版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF未由于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不错，质量低下，你也可以选择狠狠澎腕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不靠谁得到了知识，所以请不要得到书籍后就觉得占占自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF未由于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不错，质量低下，你也可以选择狠狠澎腕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不靠谁得到了知识，所以请不要得到书籍后就觉得占占自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、论文、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都在帮助客户利用科技变革的潮流。随时代变化而不断发展的O'Reilly为软件开发人员和科技商务人群创造学习型体验，使其拥有成功所需的专业技能和深刻见解。

O'Reilly的在线学习平台会员每年都可以使用数以千计的图书视频资料和交互式课程，其中的专家和创新者通过文字、视频和现场讨论来帮助会员们学习。O'Reilly的技术会议和峰会聚集了全球最顶级的开发商和关键领袖，共同碰撞出许多新的灵感，并为技术人士提供与同行面对面交流的机会。O'Reilly还为初创公司提供技术创业计划（Technology Partner Program），以促进最前沿技术和商业的发展。

业界评论

“O'Reilly Radar总是让我们学习时，学习，再学习。”
—Wired

“O'Reilly在技术一点到（真希望当初我也用了）学习资源这方面有着不可思议的品质。”
—Business 2.0

“O'Reilly Conferences是聚集了来自对其感兴趣的领域的开发者们。”
—CNN

“一直以来O'Reilly在我们心中印象，要看懂，就看书。”
—Irish Times

“Tim是位很有远见的行家。而且他投资于那些对别人来说显得很愚蠢的东西。这种远见的能力，让他在这个行业中成为超乎一般的人物。而且现在几乎所有的书都在这样做。作者与观点领域，”
—Linux Journal

献给以这本书带给我支持的人和N。
当行走或无事可思或对行时，或无它们。

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF未由于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不错，质量低下，你也可以选择狠狠澎腕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不靠谁得到了知识，所以请不要得到书籍后就觉得占占自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

目录

前言

嘿，欢迎阅读本书！很高兴你已经拿起这本书开始阅读，非常期待在接下来的 6 章中和你一起探索关于 Go 语言并发编程的主题。

Go 语言是一种美妙的语言。当它被创造并首次公开的时候，我带着极大的兴趣探索它：简洁、编译速度飞快、运行稳定、支持鸭子类型（duck typing），让我高兴的是，它原生支持并发。当我第一次使用“go 关键字”创建一个 goroutine 的时候，（我保证）我开心得只剩傻笑了。我曾经用其他一些编程语言写过并发程序，但我从未用过像 Go 语言这样这么容易实现并发的语言（我并不是说其他有这种特性的语言不存在，只是我没用过）。我已经找到了我的 Go 语言最佳实践。

在过去的几年里，我用 Go 语言写个人的脚本和项目，直到发现自己已经可以在成千上万行代码的项目中畅游。随着语言的不断发展和社区的不断壮大，我们一起找到了 Go 语言并发编程的最佳实践。一些人就他们找到的模式进行讨论，但在社区里还没有如何使用 Go 语言并发编程的综合指南。

正是考虑到这一点，我才决定写这本书。我希望可以让社区了解到关于 Go 语言并发编程的一些全面且高质量的信息：如何使用，最佳实践，以及如何将它集成到你的系统中，还有它背后的工作原理。我竭尽全力均衡这些关注点。

我希望这是一本对你有益的书。

本书的读者对象

这本书适合已经了解 Go 语言，并有一些开发经验的人。我没有解释 Go 语言的基本语法。最好了解一些其他语言的并发编程，当然这并不是必须的。

通过本书，我们将会讨论整个 Go 语言并发的技术栈：常见的并发陷阱，Go 语言并发设计原理，Go 语言并发原语中的基础语法，常见的并发模式，并发模式的设计，各种工具的使用。

由于主题涵盖的范围较广，本书适合各个使用方向的读者。下面一节将会帮助你快速找到你想要了解的内容。

本书内容

当我阅读技术书籍的时候，通常会先读能引起我兴趣的地方。或者，当我努力研究工作上所需的新技术的时候，我会先看与我工作相关的内容。无论你的出发点是什么，这里有一份关于本书的线路图，它能帮助你找到你需要的内容。

第 1 章 并发概述

本章将提供一个广泛的历史视角来说明并发的重要性，还会讨论一些并发中难以纠正的问题。它还简要介绍了 Go 语言如何帮助你解决这些问题。

如果你有并发的相关知识，或者只是想了解如何使用 Go 语言的并发原语，可以跨过此章节。

第 2 章 对你的代码建模：通信顺序进程

本章论述了推动 Go 语言设计的一些激励因素。这将帮助你在社区中与其他人顺畅交流，并理解为什么要按照 Go 语言的设计模式来编程。

第 3 章 Go 语言并发组件

在这里我们将开始深入 Go 语言并发原语的语法。我们还会介绍控制内存

访问同步的 sync 包。如果之前你从未使用 Go 语言做过并发编程，并且你正在研究如何正确地使用它，那么你应该从这里开始阅读。

通过基础的 Go 语言并发代码片段，与其他语言的并发模型做比较。严格地说，这些知识并不是必需的，但是这些概念有助于你完全理解 Go 语言的并发实现。

第 4 章 Go 语言的并发模式

在本章中，我们将开始研究如何将 Go 语言原生函数构造成合理的模式。这些模式可以解决并避免我们在组合原生函数时可能遇到的问题。

如果你已经开始写 Go 语言的并发代码了，那么这一章还是有点用的。

第 5 章 大规模并发

在这一章中，我们将组合之前学到模式，设计更合理的模型，应用到大型程序、服务和分布式系统中。

第 6 章 goroutine 和 Go 语言运行时

本章描述 Go 语言运行时如何处理调度 goroutine。这些内容主要是给那些想了解 Go 语言运行时内部构造的人学习的。

在线资源

Go 语言拥有一个非常活跃且富有激情的社区！对于那些 Go 语言的初学者，请放心，这个社区很容易找到友好、乐于助人的朋友来帮助你走上 Go 语言编程之路。下面是我最喜欢的一些面向社区的用来阅读、获得帮助，以及和你的 Gopher 伙伴进行互助的资源。

- *https://golang.org/*
- *https://golang.org/play*
- *https://go.googlesource.com/go*

- *https://groups.google.com/group/golang-nuts*
- *https://github.com/golang/go/wiki*

本书约定

本书使用以下排版约定：

斜体

表示新术语、URL、电子邮件地址、文件名和文件扩展名。

等宽字体（`constant width`）

表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

加粗等宽字体（**`constant width bold`**）

表示需要由用户输入的命令或其他文本。

等宽斜体（*`Constant width italic`*）

表示需要由用户输入的值或根据上下文确定的值替换的部分。

表示提示、建议或一般注意事项。

表示警示或告诫。

使用代码示例

本书中所包含的代码都可以在 *http://katherine.cox-buday.com/concurrency-in-go* 页面找到。所有代码都在 MIT 许可下发布的，并且可以在许可的条件下使用。

O'Reilly Safari

Safari（前身为 Safari Books Online）是一个会员制的培训、参考网站，服务于企业、政府、教育者和个人。

会员可以访问数千书籍、培训视频、学习路径、交互教程和超过 250 家出版商的企划列表，包括 O'Reilly Media、Harvard Business Review、Prentice Hall Professional、 Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 等。

更多信息请访问 *http://oreilly.com/safari*。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）
奥莱利技术咨询（北京）有限公司

我们为本书准备了一个网页，用于陈列勘误、示例代码，以及其他信息。本书的网址是：*http://bit.ly/concurrency-in-go*。

对于本书的评论和技术性问题，请发送电子邮件到：*bookquestions@oreilly.com*。

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：*http://www.oreilly.com*。

我们的 Facebook：*http://facebook.com/oreilly*。

我们的 Twitter：*http://twitter.com/oreillymedia*。

我们的 YouTube：*http://www.youtube.com/oreillymedia*。

致谢

编写本书是一个充满挑战且令人畏惧的工作。如果没有一群人支持我，审查内容，编写工具和回答问题，本书是不可能完成的。我非常感谢所有提供帮助的人，我们共同完成了本书！

一燕不成夏……

——Proverb

- Alan Donovan，他帮助我完成最初的提议，并帮助我踏上征程。
- Andrew Wilkins，我非常有幸可以在 Canonical 和他一起工作，他的洞察力、专业精神和智慧影响了这本书，他的评论使本书变得更好。
- Ara Pulido，帮助我用一个新 gopher 的视角审视本书。
- Dawn Schanafelt，我的编辑。在使本书读起来尽可能清晰的工作中起到了非常大的作用。我特别感谢她（和 O'Reilly），我写这本书时在生活的道路上遇到一些困难时给予我的耐心。

- Francesc Campoy，经常提醒我要总是注意新的 gopher。
- Ivan Daniluk，他对细节的关注和并发的兴趣确保这是一本全面而有用的书。
- Yasushi Shoji，`org-asciidoc` 的作者，`org-asciidoc` 是我用来将 Org 文档转化为 AsciiDoc 文档的工具，他并不知道他在帮助我写一本书，但是他对 bug 修复以及回答问题十分热情。
- Go 语言维护者：感谢你们的奉献。
- Org 模式的维护者，本书是使用 GNU Emacs 模式编写的，我这辈子都在使用 org，感谢大家。
- GNU Emasc 的维护者，编写本书所使用的文本编辑器，我想象不到一个对于我的生命更有意义的工具了。
- 圣路易斯公共图书馆，进行本书大部分内容写作的地方。

第 1 章

并发概述

并发是一个有趣的词，在我们的认知中，不同的人对它有不同的理解。除了“并发” 之外，你可能已经听到了被到处传播的“异步”“并行” 或 “线程” 这几个词。有些人认为这些词表示同一件事，有些人则非常具体地描绘出这些词之间的区别。如果我们要花一整本书的时间讨论并发，那么有必要先花些时间讨论一下我们所说的 “并发” 究竟是什么。

第 2 章我们将花些时间讲并发的哲学，但现在先给并发一个较为实际的定义，它将作为我们理解的基础。

当大多数人使用“并发” 这个词时，通常指的是与一个或多个进程同时发生的过程。通常，这也意味着所有这些进程都在同一时间取得进展。理解这个定义的一个简单方法是想象成人。你目前正在阅读这本书，而世界上的其他人同时也在生活。他们与你同时存在。

并发是计算机科学中的一个宽泛的话题，从这个定义中可以学到各种各样的主题：理论，建模的方法， 逻辑的正确性，实际的问题，甚至是理论物理！在本书中，我们将讨论一些辅助主题，但我们将主要基于 Go 语言环境讨论一些并发的实际问题，具体来说：如何选择模型并发，从这个模型中产生哪些问题，以及我们如何在这个模型中组合原语来解决问题。

在本章中，我们将深入了解并发成为计算机科学中的一个重要话题的原因，为什么并发是困难的，并且值得仔细研究，最重要的是，尽管这些想法富有挑战性，Go 语言可以通过其并发原语使程序更加清晰和迅捷。

就像大多数的理解事物的路径一样，我们将从一些历史开始。让我们先来看看并发是如何成为这样一个重要的主题。

摩尔定律，Web Scale 和我们所陷入的混乱

在 1965 年，戈登·摩尔写了一篇三页的论文，描述了电子市场对集成电路的整合，以及至少十年内每年集成电路中的元件数量的就会翻番。在 1975 年，他修改了这一预测，指出集成电路上的组件数每两年就会翻一番。直到最近，大约 2012 年这一预测或多或少被证实了。

一些公司预见到摩尔定律预测的增速将会放缓，并开始研究提高计算能力的替代方法。俗话说，必要性是创新之母，因此，多核处理器就这样诞生了。

这看起来像是一个巧妙的方法来解决摩尔定律的边界问题，但计算机科学家很快发现自己面临着另一条定律的限制——Amdahl 定律，以计算机架构师 Gene Amdahl 命名。

Amdahl 定律描述了一种方法，使用并行方式可以解决的问题，可以使用这种方法对其潜在的性能收益进行建模。简而言之，它指出，收益的限制取决于有多少程序必须以顺序的方式编写。

例如，假设你正在编写一个主要基于 GUI 的程序：一个用户会得到一个界面，单击一些按钮，然后发生一些事情。这类程序受限于流程中一个巨大的顺序部分：人类的交互。无论你为此程序提供多少内核，它都将始终受用户与界面交互速度的限制。

现在考虑一个不同的例子，计算 pi。它可以很容易地被划分为并行任务，这有赖于一类叫做 spigot 的算法，这个问题被称为“令人尴尬的并行问题”，这是一个技术术语，尽管听起来像是编造的。在这种情况下，可以通过将更多的核心提供给你的程序来取得显著的收益，之后困扰你的问题将变成如何组合和存储结果。

Amdahl 定律可以帮助我们理解这两个问题之间的区别，并帮助我们确定并行化是否是解决系统中的性能问题的正确方法。

对于“令人尴尬的并行问题”，建议你编写程序，使其可以水平缩放。这意味着你可以独立执行程序的实例，在多个的 CPU 或计算机上运行它，这将使系统运行时得到改善。“令人尴尬的并行问题”非常适合这个模型，因为它很容易以这样的方式构造你的程序，你可以将问题拆分开，发送到应用程序的不同实例。

在 21 世纪初，当一个新的范式开始出现时，横向缩放变得更加容易：云计算。尽管有迹象表明这个说法早在 20 世纪 70 年代就已经被使用过，但在 21 世纪初期，这个想法在时代思潮中根深蒂固。云计算隐含了一种新的规模化的方法，用于应用程序部署和水平扩展。不需要你精心策划、安装软件和维护的机器，云计算意味着可以访问大量的资源池，它们被自动调配到适当的机器中以满足工作负载需求。物理机几乎变成了短暂性的东西，并配置成某种形式以适配即将运行的程序。通常 （但不总是）这些资源池是由其他公司的数据中心托管。

这种变化催生了一种新的思维方式。突然之间，开发者获得了大量的计算能力，可以用来解决庞大的问题。解决方案现在可以分散到许多机器上甚至是全球不同的区域。云计算使得一种全新的解决方案成为可能，而这些问题之前只有技术巨头才可能解决。

但是云计算也带来了许多新的挑战。提供这些资源，在机器实例之间进行通信，聚合和存储结果都成为了需要解决的问题。但其中最困难的是如何设计并行计算的模型。你的解决方案的各部分可以在不同的机器上运行，这一事实加剧了在建模并行问题时常面临的一些问题。很快就产生了一种新型的软件（即 Web Scale）成功解决这些问题。

如果软件是 Web Scale 的，不用多想，你几乎可以确定，这一定是“令人尴尬的并行问题”。即，Web Scale 软件通常可以通过添加更多应用程序实例来

处理数十万（或更多）并发的工作负载。这产生了各种特性，如滚动升级，弹性水平可伸缩的体系结构，以及基于地理的分布式架构。它还引入了新的复杂程度，包括理解和容错。

因此，在这个多核、云计算、Web Scale 以及可并行化或不可并行化的问题的世界中，我们发现现代开发者可能有些不知所措。众所周知，增速放缓已经开始影响到我们，我们预计将面临在现在所用的硬件范围内解决未来问题的挑战。在 2005 年，Herb Sutter 创作了一篇写给 Dobb 博士的文章“The free lunch is over: A fundamental turn toward concurrency in software”。标题很贴切，文章也有先见之明。在文章的最后 Sutter 指出，“我们迫切需要一个比现有语言提供的层次更高的并发编程模型。”

想理解为什么 Sutter 使用如此强烈的言语，我们必须看看为什么并发是如此难以正确地构建。

为什么并发很难？

众所周知，并发代码是很难正确构建的。它通常需要完成几个迭代才能让它按预期的方式工作，即使这样，在某些时间点（更高的磁盘利用率、更多的用户登录到系统等）到达之前， bug 在代码中存在数年的事情也不少见，以至于以前未被发现的 bug 在后面显露出来。事实上，对于本书，我已经尽可能多地反复查看了代码，以尝试和减轻这一点。

幸运的是，在使用并发编程时，每个人都会遇到同样的问题。正因为如此，计算机科学家已经对共同的问题进行了标记，这使我们能够讨论它们是如何产生的，为什么会这样，以及如何解决它们。

那让我们开始吧。下面是一些最常见的问题，它们让使用并发代码既令人沮丧又让人感到有趣。

竞争条件

当两个或多个操作必须按正确的顺序执行，而程序并未保证这个顺序，就会发生竞争条件。

大多数情况下，这将在所谓的数据中出现，其中一个并发操作尝试读取一个变量，而在某个不确定的时间，另一个并发操作试图写入同一个变量。

下面是一个基本示例：

```
1.  var data int
2.  go func() {  ❶
3.      data++
4.  }()
5.  if data==0 {
6.      fmt.Printf("the value is %v.\n", data)
7.  }
```

❶ 在 Go 语言中，你可以使用关键字 go 来并发地运行一个函数。这样做会产生所谓的 *goroutine*。我们将在第 3 章的“goroutine”一节中详细讨论这一点。

在这里，第 3 行和第 5 行都试图访问变量 data，但并不能保证以什么顺序进行访问。运行这段代码有三种可能的结果：

- 不打印任何东西。在这种情况下，第 3 行在第 5 行之前执行。
- 打印“the value is 0”。在这种情况下，第 5 行 和第 6 行在第 3 行之前执行。
- 打印“the value is 1”。在这种情况下，第 5 行在第 3 行之前执行，但第 3 行在第 6 行之前执行。

如你所见，虽然只有几行不正确的代码，但在你的程序中引入了巨大的不确定性。

在大多数情况下，引入数据竞争的原因是因为开发人员在用顺序性的思维来思考问题。他们假设，某一行代码在另一个之前就会先运行。他们认为上述 goroutine 将在 `if` 语句读取数据变量之前被安排执行。

在编写并发代码时，你必须仔细地遍历所有可能的方案。除非你正在使用本书稍后将介绍的一些技术，否则你无法保证你的代码将按照它在源代码中的顺序运行。我发现，有时候想象在两个操作之间会经过很长一段时间很有帮助。假设调用 goroutine 的时间和它运行的时间相差 1h。那程序的其余部分将如何运行呢？如果在 goroutine 执行成功和程序执行到 `if` 语句之间也花费了一小时，又会发生什么呢？以这种方式思考对我很有帮助，因为对于计算机来说，规模可能不同，但相对的时间差异或多或少是相同的。

实际上，有些开发人员在他们的代码中使用了很多这种休眠语句，因为它似乎解决了他们的并发问题。让我们在前面的程序中尝试这样修改一下：

```
var data int
go func() { data++ }()
time.Sleep(1*time.Second) // 这种方式不优雅
if data == 0 {
     fmt.Printf("the value is %v.\n" data)
}
```

我们的数据竞争问题解决了吗？并没有。事实上，在这个程序中之前的三个结果仍然有可能出现，只是可能性更小了。我们在调用 goroutine 和检查数据值之间的休眠的时间越长，我们的程序就越接近正确，但那只是概率上接近逻辑的正确性；它永远不会真的变成逻辑上的正确。

除此之外，这让我们的算法变得低效。我们现在不得不休眠 1s，来降低我们的程序出现数据竞争的可能。但如果我们使用正确的工具，我们就不必等待了，或者等待时间可能只有 1μs。

这里的结论是，你应该始终以逻辑正确性为目标。在代码中引入休眠可以方便地调试并发程序，但这并不能称之为一个解决方案。

竞争条件是最难以发现的并发 bug 类型之一，因为它们可能在代码投入生产多年之后才出现。通常代码正在执行时环境产生变化，或发生了某些罕见的事情，都有可能使其浮现出来。往往代码只是看上去在用正确的方式来执行，但是事实上只是执行的顺序是正确的这件事本身的概率比较大而已，最终早晚有可能会出现一些意想之外的结果。

原子性

当某些东西被认为是原子的，或者具有原子性的时候，这意味着在它运行的环境中，它是不可分割的或不可中断的。

那么这到底意味着什么，为什么在使用并发代码时知道这一点很重要？

第一件非常重要的事情是“上下文（context）”这个词。可能在某个上下文中有些东西是原子性的，而在另一个上下文中却不是。在你的进程上下文中进行原子操作在操作系统的上下文中可能就不是原子操作；在操作系统环境中原子操作在机器环境中可能就不是原子的，在你的机器上下文中原子操作在你的应用程序的上下文中可能不是原子的。换句话说，操作的原子性可以根据当前定义的范围而改变。这种特性对你来说有利有弊！

在考虑原子性时，经常第一件需要做的事就是定义上下文或范围，然后再考虑这些操作是否是原子性的。一切都应当遵循这个原则。

有趣的事实

2006 年，游戏公司 Blizzard 成功起诉了 MDY Industries 索赔 600 万美元，因为该公司制作了一个名为“Glider”的程序，该程序可以在没有用户干预的情况下自动操作他们的游戏“魔兽世界”。这些类型的程序通常被称为“bots”（机器人的缩写）。

当时，“魔兽世界”有一个反作弊程序叫做“Warden”，它可以在你进行游戏期间任意地运行。除此之外，Warden还会扫描主机的内存并进行启发式扫描，寻找可能用于作弊的程序。

Glider利用原子上下文的概念成功避过了这种检查。Warden认为扫描机器上的内存是一种原子操作，但在开始扫描之前，Glider利用硬件中断来隐藏自己！Warden认为对内存的扫描在进程的上下文中是原子的，而不是在操作系统的上下文中。

现在让我们来看一下术语“不可分割”（indivisible）和“不可中断”（uninterruptible）。这些术语意味着在你所定义的上下文中，原子的东西将被完整的运行，而在这种情况下不会同时发生任何事情。这仍然是一个整体，所以我们来看一个例子：

```
i++
```

这是一个任何人都可以设计的简单例子，但它很容易证明原子性的概念。它可能看起来很原子，但是简要地分析一下就会发现其中有以下步骤：

- 检索i的值。
- 增加i的值。
- 存储i的值。

尽管这些操作中的每一个都是原子的，但三者的结合就可能不是，这取决于你的上下文。这揭示了原子操作的一个有趣的性质：将它们结合并不一定会产生更大的原子操作。使一个操作变为原子操作取决于你想让它在哪个上下文中。如果你的上下文是一个没有并发进程的程序，那么该代码在该上下文中就是原子的。如果你的上下文是一个goroutine，它不会将i暴露给其他goroutine，那么这个代码就是原子的。

为什么我们要关心这些呢？原子性非常重要，因为如果某个东西是原子的，

隐含的意思是它在并发环境中是安全的。这使我们能够编写逻辑上正确的程序，而且稍后将会看到，这甚至可以作为优化并发程序的一种方式。

大多数语句不是原子的，更不用说函数、方法和程序了。如果原子性是构成逻辑正确程序的关键，而大多数语句不是原子的，那么我们如何调和这个矛盾呢？稍后我们会进一步讨论，但是，总之我们可以通过各种技术来使操作强制保持原子性。之后的工作就是确定代码的哪些部分需要是原子化，以及在什么粒度级别上原子化。我们将在下一节讨论其中的一些挑战。

内存访问同步

假设有这样一个数据竞争：两个并发进程试图访问相同的内存区域，它们访问内存的方式不是原子的。之前有一个简单的数据竞争的例子，这里稍作修改就可以说明：

```
var data int
go func() { data++ }()
if data==0 {
    fmt.Println("the value is 0.")
} else {
    fmt.Printf("the value is %v.\n", data)
}
```

我们在这里添加了一个 `else` 子句，所以不管数据的值如何，我们总会得到一些输出。请记住，正如之前所介绍，如果有一个数据竞争存在，那么该程序的输出将是完全不确定的。

实际上，程序中需要独占访问共享资源的部分有一个专有名词，叫临界区（critical section）。在这个例子中，我们有三个临界区：

- 我们的 goroutine 正在增加数据变量。
- 我们的 `if` 语句，它检查数据的值是否为 0。
- 我们的 `fmt.Printf` 语句，在检索并输出数据的值。

有很多方法可以保护你的程序的临界区，Go 语言在设计时有一些更好的想法来解决这个问题，不过解决这个问题的其中一个办法是在你的临界区之间内存访问做同步。我们来看看是什么样的。

下面的代码不是 Go 语言中惯用的方法（我不建议像这样解决你的数据竞争问题），但它很简单地演示了内存访问同步。如果这个例子中的任何类型，函数或者方法对你来说是陌生的，那没关系。借助后面的备注关注内存访问同步的概念就行。

```
var memoryAccess sync.Mutex ❶
var value int
go func() {
    memoryAccess.Lock() ❷
    value++
    memoryAccess.Unlock() ❸
}()
memoryAccess.Lock() ❹
if value == 0 {
    fmt.Printf("the value is %v.\n", value)
}else{
    fmt.Printf("the value is %v.\n", value)
}
memoryAccess.Unlock() ❺
```

❶ 在这里，我们添加一个变量，它将允许我们的代码对内存数据的访问做同步。我们将在第 3 章的“sync 包”中详细介绍 sync.Mutex 类型。

❷ 在这里，我们声明，直到我们的声明结束前，我们的 goroutine 应该独占该内存的访问权。

❸ 在这里，我们宣布 goroutine 使用完了这段内存。

❹ 在这里，我们再次声明以下条件语句应独占变量 data 内存的访问权。

❺ 在这里，我们宣布我们的独占结束。

在这个例子中，我们为开发者制定了一个约定。无论何时，开发人员想要访问变量 data 的内存，必须首先调用 Lock，当访问结束后，他们必须调用 Unlock。我们可以假设这两条语句之间的代码拥有对变量 data 的独占访问权；至此，我们已经成功地对内存的访问进行了同步。另外请注意，如果开发者不遵循这个约定，我们就无法保证独占访问！我们将在第 4 章的“约束”中再回顾这一思想。

你可能已经注意到，虽然我们已经解决了数据竞争，但并没有真正解决我们的竞争条件！这个程序的操作顺序仍然是不确定的。我们刚刚只是缩小了非确定性的范围。在这个例子中，goroutine 和 `if` 和 `else` 块都有可能会先执行。我们仍然不知道在这个程序的执行过程中哪段代码会先执行。之后，我们将探索正确解决这类问题的方法。

从表面上看，这似乎很简单：如果你发现你的代码中有临界区，那就添加锁来同步内存访问！很简单，对吧？那么……还有些问题。

的确，你可以通过内存访问同步来解决一些问题，但正如我们刚刚看到的，它不会自动解决数据竞争或逻辑正确性问题。此外，它也可能造成维护和性能问题。

请注意，我们之前创建了一个约定，当需要对某些内存进行独占访问时要进行声明。约定是好的，但它们也容易被忽略，特别是在软件工程中，业务需求有时有时间限制。通过这种方式同步对内存的访问，需要期望所有开发人员在当前和未来都遵循相同约定。这是一个非常艰巨的任务。值得庆幸的是，在这本书的稍后部分，我们还将探讨一些方法，可以有效地帮助我们的同事。

以这种方式同步对内存的访问还有性能上的问题。更多的细节我们将在第 3 章的“sync 包”中描述，不过现在你就能看到 `Lock` 的调用会使我们的程序变慢。每次执行这些操作时，我们的程序就会暂停一段时间。

这带来了两个问题：

- 我的临界区是否是频繁进入和退出？
- 我的临界区应该有多大？

在程序的上下文中解决这两个问题是一种艺术，并且增加了内存访问同步的难度。

同步对内存的访问在对并发进行建模时还存在一些问题，我们将在下一节讨论这些问题。

死锁、活锁和饥饿

前面的部分都是关于程序正确性的讨论，如果这些问题得到正确的处理，你的程序将永远不会给出错误的答案。不幸的是，即使你成功处理了这些问题，还有另一类问题需要解决：死锁、活锁和饥饿。所有这些问题都与你的程序密切相关，它们保证了你的程序在任何时候都在做着一些真正有用的事。如果没有正确处理，程序可能会进入一种完全停止正常工作的状态。

死锁

死锁程序是所有并发进程彼此等待的程序。在这种情况下，如果没有外界的干预，这个程序将永远无法恢复。

这听起来很严峻，那是因为的确如此！ Go 语言的运行时会尽其所能，检测一些死锁（所有的 goroutine 必须被阻塞，或者“asleep”[注1]），但是这对于防止死锁并没有太多的帮助。

为了帮助理解死锁是什么，我们先来看一个例子。同样，忽略任何你不知道的类型，函数，方法或是你不知道的包，只理解什么是死锁即可。

```
type value struct {
    mu sync.Mutex
    value int
}

var wg sync.WaitGroup
printSum := func(v1, v2 *value) {
    defer wg.Done()
    v1.mu.Lock() ❶
```

注 1：有一个被接受的建议是允许运行时检测部分死锁，但是还没有实现。有关更多信息，请参见 *https://github.com/golang/go/issues/13759*。

```
    defer v1.mu.Unlock()❷

    time.Sleep(2*time.Second) ❸
    v2.mu.Lock()
    defer v2.mu.Unlock()

    fmt.Printf("sum=%v\n", v1.value + v2.value)
}

var a, b value
wg.Add(2)
go printSum(&a, &b)
go printSum(&b, &a)
wg.Wait()
```

❶ 在这里，我们尝试进入临界区来传入一个值。

❷ 在这里，我们使用 defer 语句在 printSum 返回之前退出临界区。

❸ 在这里，我们休眠了一段时间来模拟一些工作（并触发死锁）。

如果尝试运行此代码，你可能会看到：

```
fatal error: all goroutines are asleep - deadlock!
```

为什么呢？ 如果仔细观察，你将在此代码中看到时机问题。以下是运行时的图形表示。这些框表示函数，水平线表示调用这些函数，竖线表示图形头部的函数生存时间（见图 1-1）。

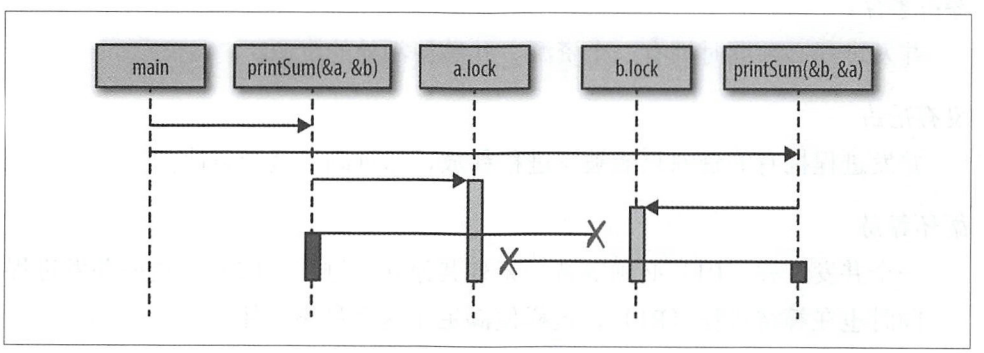


图 1-1：一个因时间问题导致死锁的演示

本质上，我们创建了两个不能转动的齿轮：第一次调用 printSum 锁定 a，然后试图锁定 b，但在此期间，第二次调用 printSum 已锁定 b 并试图锁定 a。这两个 goroutine 都无限地等待着。

讽刺

为了保持这个例子简单，我使用一个 `time.Sleep` 来触发死锁。但是，这引入了一个竞争条件！你能找到吗？

一个逻辑上“完美”的死锁将需要正确地同步[注2]。

以图形的方式展示为什么会出现死锁似乎很明确，但是更严格的定义会给我们带来更多的好处。事实证明，出现死锁有几个必要条件。1971 年，Edgar Coffman 在一篇论文中列举了这些条件。这些条件现在被称为 Coffman 条件，是帮助检测、防止和纠正死锁的技术依据。

Coffman 条件如下：

相互排斥

并发进程同时拥有资源的独占权。

等待条件

并发进程必须同时拥有一个资源，并等待额外的资源。

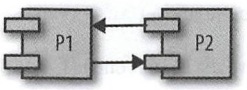
没有抢占

并发进程拥有的资源只能被该进程释放，即可满足这个条件。

循环等待

一个并发进程（P1）必须等待一系列其他并发进程（P2），这些并发进程同时也在等待进程（P1），这样便满足了这个最终条件。

注 2：我们实际上并不能保证 goroutines 的运行顺序，或者需要多长时间才能启动。虽然不太可能，但是一个 goroutine 可以在另一个 goroutine 开始之前获得和释放锁，从而避免死锁，这是有道理的。



来看看我们设计的程序，并确定它是否符合所有四个条件：

1. printSum 函数确实需要 a 和 b 的独占权，所以它满足了这个条件。
2. 因为 printSum 持有 a 或 b 并正在等待另一个，所以它满足这个条件。
3. 我们没有任何办法让我们的 goroutine 被抢占。
4. 我们第一次调用 printSum 正在等待我们的第二次调用；反之亦然。

是的，我们写出来的无疑是一个死锁了。

这些规则也帮助我们防止死锁。如果确保至少有一个条件不成立，我们可以防止发生死锁。不幸的是，实际上这些条件很难推理，因此很难预防。网络上散布着诸如你和我这样的开发者的疑问，他们想知道为什么一小段代码是死锁的。通常情况下，一旦有人指出，这是非常明显的，但往往需要另一双眼睛。我们将在本章后面的“确定并发安全性”中讨论这是为什么。

活锁

活锁是正在主动执行并发操作的程序，但是这些操作无法向前推进程序的状态。

你曾经在走廊走向另一个人吗？她移动到一边让你通过，但你也做了同样的事情。所以你转到另一边，但她也是这样做的。想象一下这个情形永远持续下去，你就明白了活锁。

我们实际上编写一些代码来演示这种情况。首先，我们将设置一些辅助函数来简化示例。为了有一个可以工作的例子，这里的代码利用了我们尚未涉及的几个主题。我不建议试图了解它的细节，直到你有把握可以使用好 sync 包。相反，我建议遵循代码标注来理解强调的部分，然后将注意力转向包含示例核心的第二个代码块。

```
cadence := sync.NewCond(&sync.Mutex{})
go func() {
    for range time.Tick(1*time.Millisecond) {
        cadence.Broadcast()
    }
}()

takeStep := func() {
    cadence.L.Lock()
    cadence.Wait()
    cadence.L.Unlock()
}

tryDir := func(dirName string, dir *int32, out *bytes.Buffer) bool {❶
    fmt.Fprintf(out, " %v", dirName)
    atomic.AddInt32(dir, 1) ❷
    takeStep() ❸
    if atomic.LoadInt32(dir) == 1 {
        fmt.Fprint(out, ". Success!")
        return true
    }

    takeStep()
    atomic.AddInt32(dir, -1) ❹
    return false
}

var left, right int32
tryLeft := func(out *bytes.Buffer) bool { return tryDir("left", &left, out) }
tryRight := func(out *bytes.Buffer) bool { return tryDir("right", &right, out) }
```

❶ tryDir 允许一个人尝试向一个方向移动，并返回是否成功。dir，每个方向都表示为试图朝这个方向移动的人数。

❷ 首先，我们宣布将要向这个方向移动一个距离。我们将在第 3 章中详细讨论这个 atomic 包。现在，你只需要知道这个包的操作是原子操作。

❸ 为了演示活锁，每个人都必须以相同的速度或节奏移动。takeStep 模拟所有对象之间的一个不变的节奏。

❹ 这里的人意识到他们不能向这个方向走而放弃。我们通过把这个方向减 1 来表示。

```
walk := func(walking *sync.WaitGroup, name string) {
    var out bytes.Buffer
    defer func() { fmt.Println(out.String()) }()
    defer walking.Done()
    fmt.Fprintf(&out, "%v is trying to scoot:", name)
```

```
    for i:=0; i<5; i++ { ❶
        if tryLeft(&out) || tryRight(&out) { ❷
            return
        }
    }
    fmt.Fprintf(&out, "\n%v tosses her hands up in exasperation!", name)
}
var peopleInHallway sync.WaitGroup ❸
peopleInHallway.Add(2)
go walk(&peopleInHallway, "Alice")
go walk(&peopleInHallway, "Barbara")
peopleInHallway.Wait()
```

❶ 我对尝试次数进行了人为限制，以便此程序能结束。在一个有活锁的程序中，可能没有这个限制，这就是为什么它是一个问题！

❷ 首先，这个人会试图向左走，如果失败了，他们会尝试向右走。

❸ 这个变量为程序提供了一个等待直到两个人都能够相互通过或放弃的方式。

输出如下：

```
Alice is trying to scoot: left right left right left right left right left right
Alice tosses her hands up in exasperation!
Barbara is trying to scoot: left right left right left right left right
left right
Barbara tosses her hands up in exasperation!
```

你可以看到，Alice 和 Barbara 在最终退出之前，会持续竞争。

这个例子演示了使用活锁的一个十分常见的原因：两个或两个以上的并发进程试图在没有协调的情况下防止死锁。这就好比，如果走廊里的人都同意，只有一个人会移动，那就不会有活锁；一个人会站着不动，另一个人会移到另一边，他们就会继续移动。

在我看来，活锁要比死锁更复杂，因为它看起来程序好像在工作。如果一个活锁程序在你的机器上运行，那你可以通过查看 CPU 利用率来确定它是否在做处理某些逻辑，你可能会认为它确实是在工作。根据活锁的不同，它甚至可能发出其他信号，让你认为它在工作。然而，你的程序将会一直上演“hallway-shuffle”的循环游戏。

活锁是一组被称为“饥饿”的更大问题的子集。我们下一节会讲到。

饥饿

饥饿是在任何情况下，并发进程都无法获得执行工作所需的所有资源。

当我们讨论活锁时，每个 goroutine 的资源是一个共享锁。

活锁保证讨论与饥饿是无关的，因为在活锁中，所有并发进程都是相同的，并且没有完成工作。更广泛地说，饥饿通常意味着有一个或多个贪婪的并发进程，它们不公平地阻止一个或多个并发进程，以尽可能有效地完成工作，或者阻止全部并发进程。

这里有一个程序的例子，有一个贪婪的 goroutine 和一个平和的 goroutine：

```
var wg sync.WaitGroup
var sharedLock sync.Mutex
const runtime = 1*time.Second

greedyWorker := func() {
    defer wg.Done()

    var count int
    for begin := time.Now(); time.Since(begin) <= runtime; {
        sharedLock.Lock()
        time.Sleep(3*time.Nanosecond)
        sharedLock.Unlock()
        count++
    }
    fmt.Printf("Greedy worker was able to execute %v work loops\n", count)
}

politeWorker := func() {
    defer wg.Done()

    var count int
    for begin := time.Now(); time.Since(begin) <= runtime; {
        sharedLock.Lock()
        time.Sleep(1*time.Nanosecond)
        sharedLock.Unlock()
```


```
            sharedLock.Lock()
            time.Sleep(1*time.Nanosecond)
            sharedLock.Unlock()

            sharedLock.Lock()
            time.Sleep(1*time.Nanosecond)
            sharedLock.Unlock()

            count++
        }
        fmt.Printf("Polite worker was able to execute %v work loops.\n", count)
    }

    wg.Add(2)
    go greedyWorker()
    go politeWorker()

    wg.Wait()
```

输出如下：

```
Polite worker was able to execute 289777 work loops.
Greedy worker was able to execute 471287 work loops.
```

贪婪的 worker 会贪婪地抢占共享锁，以完成整个工作循环，而平和的 worker 则试图只在需要时锁定。两种 worker 都做同样多的模拟工作（sleeping 时间为 3ns），但是你可以看到，在同样的时间里，贪婪的 worker 工作量几乎是平和的 worker 工作量的两倍！

假设两种 worker 都有同样大小的临界区，而不是认为贪婪的 worker 的算法更有效（或调用 `Lock` 和 `Unlock` 的时候，它们也不是缓慢的），我们得出这样的结论，贪婪的 worker 不必要地扩大其持有共享锁上的临界区，并阻止（通过饥饿）平和的 worker 的 goroutine 高效工作。

请注意，我们这里的技术用于识别饥饿——一个 metric。饥饿会为记录和取样提供一个很好的 metric。一个发现和解决饥饿的方法是，通过记录来确定进程工作速度是否和你预期的一样高。

找到一个平衡

值得一提的是，前面的代码示例还可以作为内存访问同步的性能影响的示例。因为同步访问内存是昂贵的，所以将我们的锁扩展到临界段之外是有利的。另一方面，这样做我们将冒着饿死其他并发进程的风险。

如果你使用了内存访问同步，你将不得不在粗粒度同步和细粒度同步之间找到一个平衡点。当需要对应用程序进行性能调优时，我强烈建议只将内存访问同步限制在关键部分；如果同步成为性能问题，你可以一直扩展范围。走另一条路要难得多。

所以，饥饿会导致你的程序表现不佳或不正确。前面的示例演示了低效场景，但是如果你有一个非常贪婪的并发进程，以至于完全阻止另一个并发进程完成工作，那么你就会遇到一个更大的问题。

我们还应该考虑到来自于外部过程的饥饿。请记住，饥饿也可以应用于CPU、内存、文件句柄、数据库连接：任何必须共享的资源都是饥饿的候选者。

确定并发安全

最后，我们来谈谈开发并发代码的最困难的地方，即所有其他问题的根源——人。每行代码的后面至少有一个人。

正如我们所发现的，并发代码很难写有很多原因。如果你是一名开发人员，并且你在引入新功能时尝试解决所有这些问题，或修复程序中的错误作出正确的修改，确实很困难。

如果你从一个空白的石板开始，需要建立一个合理的方法来模拟你的问题空间，并且涉及并发，那么很难找到合适的抽象层次。你如何向调用者说明并发？

你使用什么技术来创建易于使用和修改的解决方案？这个问题的并发性是什么水平？虽然有办法以结构化的方式思考这些问题，但它仍然是一门艺术。

作为一个与现有代码交互的开发人员，如何利用并发编写代码，以及如何安全地使用代码并不总是那么明确的。拿这个函数来说：

```
// CalculatePi 在开始和结束位置之间计算 Pi 的数字
func CalculatePi(begin, end int64, pi *Pi)
```

以较高的精度计算 Pi 是个非常棒的例子，但是这个例子引发了很多问题：

- 如何使用这个函数。
- 为了并发地调用，我是否需要多次实例化这个函数。
- 看起来我传入 Pi 后，函数的所有实例都将在当前实例上运行，那我需要内存访问同步吗？或者 Pi 类型帮我做了这个事情？

一个函数就带来了这么多问题。想象一下任何一个中等规模的程序，你就能理解并发的复杂性了。

注释可以带来一些改观。如果 CalculatePi 函数是这样写的：

```
// CalculatePi 计算 Pi 从 begin 到 end 的那一部分。
// 在内部，CalculatePi 将并发地调用 FLOOR[(end-begin)/2]。
//
// CalculatePi 会递归地调用。
// Pi 的写入同步由 Pi 结构体内部处理
func CalculatePi(begin, end int64, pi *Pi)
```

现在明白了我们可以直接调用这个函数，而不用担心并发或同步的问题。重要的是，注释涵盖了这些内容：

- 谁负责并发？
- 如何利用并发原语解决这个问题的？
- 谁负责同步？

当公开涉及并发的函数，方法和变量时，请替同事和未来的自己考虑一下：宁可让注释变得冗长，也应该尽可能涵盖这三个方面。

还要考虑到，这个函数模糊地暗示了我们对它的模型理解是错误的。也许我们应该采取一种纯函数式的方法，确保我们没有做多余的工作。

```
func CalculatePi(begin, end int64) []uint
```

这个函数的声明消除了同步的问题，但仍然有并发使用的问题。我们可以再次修改声明，抛出另一个相关的信号，反馈正在发生的事情：

```
func CalculatePi(begin, end int64) <-chan uint
```

这里我们看到了所谓channel的第一个用法。稍后将在第3章的“channel”中探讨的细节，现在你只要理解这表明`CalculatePi`将至少会启动一个goroutine，而我们不用考虑这些问题即可。

然而，这些修改会带来性能上的影响，这使我们又回到了平衡明确性和性能的问题上。明确性非常重要，因为我们希望尽可能让将来使用这些代码的人正确使用它，不过显而易见，性能也很重要重要。两者并不相互排斥，但难以协调。

现在考虑一下如何告诉别人这些障碍，并尝试在团队项目中推广你的做法。

这是一个好问题。

好消息是，Go语言已经取得了进展，使得这些类型的问题更容易解决。语言本身有利于可读性和简单性。它对并发代码进行建模的方式有助于正确性、可组合性和可伸缩性。事实上，Go语言处理并发的方式实际上可以帮助更清楚地表达问题域！让我们来看看为什么是这样的情况。

面对复杂性的简单性

以上，我描绘了一个非常严峻的画面。并发在计算机科学中当然是一个困难的领域，但是我希望给你留下一个希望：这些问题不是棘手的，并且使用 Go 语言的并发原语，可以更安全，更清晰地表达并发算法。我们讨论的运行时间和通信困难并不是 Go 语言解决的，但是它们变得非常容易。在下一章中，我们将会探讨为什么会存在“运行时与通信”问题的根源。在这里，我们花一点时间来探讨 Go 语言的并发原语实际上可以更容易地对问题域进行建模并更清楚地表达算法。

Go 语言的运行时完成了大部分的繁重工作，并为 Go 语言的大部分并发优势提供了基础。我们将讨论第 6 章如何工作，但在这里我们将讨论这些事情如何让你的工作变得更轻松。

首先讨论 Go 语言的并发，低延 GC。在一种语言中有 GC 是否是正确的，开发者之间经常会有争论。批评者认为，在需要实时性或确定性的特定领域中 GC 妨碍了正常的工作，暂停程序中的所有活动来进行 GC 简直是不可接受的。虽然有一些影响，但是 Go 语言出色的 GC 效果，大大减少了使用者对 Go 语言的 GC 工作细节的关注。从 Go 1.8 开始，GC 暂停一般在 10~100μs 之间！

这对你有什么帮助呢？内存管理可以称为计算机科学领域的另一个难题，当与并发结合使用时，编写正确的代码变得非常困难。如果大多数开发人员不需要担心只有 10μs 的暂停时间，那么通过不强迫用户管理内存，Go 语言可以更容易地使用并发程序，更不用说跨进程并发。

Go 语言的运行时也会自动处理并发操作到操作系统线程上。这是一个整体，我们将在第 3 章的“goroutine”中确切地说明这意味着什么。为了理解这对你有什么帮助，你只需要知道它可以让你直接将并发问题映射到并发结构，而不是处理启动和管理线程的细节，并在可用线程间均匀映射逻辑。

例如，假设你编写了一个 Web 服务器，并且你希望每个连接都可以被其他连接同时处理。在某些语言中，在 Web 服务器开始接受连接之前，你可能必须创建一个线程集合（通常称为线程池），然后将传入连接映射到线程。然后，在你创建的每个线程中，你需要循环该线程上的所有连接，以确保它们都获得了一些 CPU 时间。另外，你必须编写你的连接处理逻辑，使它可以与其他连接公平地共享。

相比之下，在 Go 语言中你可以编写一个函数，然后用 Go 语言关键字预先调用它。运行时会自动处理我们讨论的所有内容！当你正在经历设计你的程序的过程时，你认为在哪种模式下你更有可能达到并发？你认为哪个更可能是正确的？

Go 语言的并发原语也使得处理更大的问题变得容易。正如我们将在第 3 章的“channel”中看到的，Go 语言的 channel 原语为并发进程之间的通信提供了可组合、并发安全的方式。

我已经掩盖了这些工作的大部分细节，但是我想给你一些关于 Go 语言如何让你在程序中使用并发的感觉，以帮助你以一种清晰和高效的方式解决你的问题。在接下来的章节中，我们将讨论并发理念，以及为什么 Go 语言有这么多的正确性。如果你急于了解某些代码，你可能会想跳转到第 3 章。

第 2 章

对你的代码建模：通信顺序进程

并发与并行的区别

事实上，并发与并行的区别经常被忽视或者误解。当和开发者进行交流的时候，这两种模式总是被混用来表达“某种和其他事物同时运行的事物”。有些时候在某些环境中使用“并行”是正确的，但是，当开发者们讨论代码的时候，他们真的应该使用“并发”这个词。

如何区分这两个概念显得过于卖弄学问。并发与并行的区别在你对代码进行建模的时候是一个非常强力的抽象，而 Go 语言则充分利用了这一点。让我们来了解一下这两个概念的不同点，来理解原本抽象的力量。让我们从一个很简单的陈述开始：

> 并发属于代码；并行属于一个运行中的程序。

这是一个有趣的区别。我们不是经常把这两种事物认为是相同的吗？我们为了可以并行执行才写的代码，难道不是吗？

好吧，让我们稍微考虑一下。假设我故意写了两部分可以并行运行的程序，我可以保证在程序实际运行的时候并行执行么？我在只有一个核心的机器上运行会发生什么？你们中某些人会认为程序会并行运行，但事实并不是这样的！

代码块可能表现的像是在并发的执行，但是事实上，它们在用不可被辨别的速度进行顺序执行。CPU 的上下文在一个时间颗粒度之内一直在不同的程序之间进行切换分享 CPU 时间使得任务好像是在并行执行。如果在有两个 CPU 核心的机器上执行相同的二进制文件，代码块可能真的是在并行执行！

这揭示了一些很有趣且重要的事情。首先，我们并没有编写并行的代码，只有我们希望可以并行执行的并发代码。另外，并行是我们程序运行时的属性，而不是我们的代码。

其次，就是可能（或者这么做是可取的）对我们所写的并发代码是否真的并行执行，保持不知情。这只有在我们程序模型之下的抽象层实现：并发原语、程序的运行时、操作系统，操作系统所运行的平台（运行在 hypervisor，容器和虚拟机时），以及最终的 CPU，这些抽象给予我们区分并发与并行的能力，最终，给了我们灵活而有力的表达。我们回到这个问题本身。

第三个也是最后一个有意思的事情是并行是一个时间或者上下文的函数。还记得我们在第 1 章“原子性”中所讨论的上下文的概念吗？在那里，上下文被定义成一个操作被认为是原子性的界限。这里，上下文定义为两个或者以上的操作被认为是并行的界限。

例如，如果我们的上下文是一段 5s 的时长，我们执行了两个分别消耗 1s 钟执行的操作，我们应该认为这些操作是并行执行的。如果我们的上下文是 1s，我们应该认为操作是分别运行的。

对于我们来说，用不同的时间对上下文的概念进行重定义并不是一件好事，但是请记住上下文和时间并没有关系。我们可以把上下文定义成我们程序所在运行的进程，一个操作系统的线程，或者是一台机器。这很重要，因为你所定义的上下文是和并发性以及正确性密切相关。就像原子操作可以按照你所定义的上下文来定义是否为原子性，并发操作也依据你所定义的上下文来确定正确性。一切都是相关的。

这里介绍的内容可能有些抽象，所以我们来看一个例子。我们假设所讨论的上下文是你的计算机。把物理上的理论放在一边，我们可以合理地认为在我

的计算机上执行的一个进程，不会影响你的计算机上执行的一个进程的逻辑。如果我们都开启一个计算器进程来做一些简单的数学运算，我所操作的计算器不应该影响你所操作的计算器。

这是一个很笨的例子，但是如果我们拆分它，就能看到所有的片段：我们的计算机就是上下文，整个过程就是并发操作。在这种情况下，我们选择用独立的计算机、操作系统和进程来考虑问题，从而对并发操作进行建模。这些抽象概念让我们自信地断言正确性。

这真的是一个很笨的示例吗？

使用独立的个人计算机作为例子像是一种人为的观点，但是个人计算机并不是一直都这么随处可见的！直到 20 世纪 70 年代末，大型主机还是标准，而开发者们在思考关于并发的问题的时候的上下文经常是一个程序的进程。

现在很多开发者都在从事分布式系统相关工作，而且正在向另一个方向发展！我们现在开始从 hypervisor、容器，以及虚拟机的角度来思考我们的并发上下文。

我们可以理所当然的期待一台计算机上的一个进程不被其他计算机上的进程影响（假设它们不是同一个分布式系统的组成部分），但是，我们能期待在相同的计算机上运行的两个进程不影响另一个进程的逻辑么？进程 A 可能会覆盖某些进程 B 正在读取的文件，或者，在一个不可靠的操作系统上，进程 A 甚至可能会损坏进程 B 正在读取的内存。这么做的后果是会导致更多的漏洞被利用。

然而，在进程的角度，事情考虑起来保持着相对简单的方式。如果我们回到我们计算器的例子，也可以合理地期待两个用户在相同的机器上运行的两个计算器的进程，也应该可以认为它们对于逻辑的操作也应该是彼此独立的。

幸运的是，进程的边界线以及操作系统帮助我们按照一个符合逻辑的方式来思考这些问题。但是，我们仍然可以见到开始对并发担心的开发者，而且这个问题只会变得更加严重。

如果我们向操作系统线程的界限再前进一步的话会发生什么？所有在第 1 章“为什么并发很难”中列举的问题都开始出现：条件竞争、死锁、活锁，以及饥饿。如果我们有一个所有计算机上的用户都可见的计算器进程，就会使并发逻辑变的很难正确。我们应该已经开始担心对于内存访问同步以及给正确的用户取回正确的结果。

事实上随着我们对于抽象栈的进一步深入，如何正确的对事物进行建模变得越来越难理解，对我们来说越来越重要。相反地，抽象对于我们来说越来越重要啦。换言之，编写正确的并发逻辑越难，越需要我们将很简单的并发原语组合起来使用。不幸的是，我们行业中的大部分并发逻辑都是写在最高等级之一的操作系统线程抽象之上的。

在 Go 语言将之公开于众之前，大部分的主流编程语言都有一系列的抽象层。如果你想写并发代码，你需要对你的程序按照线程以及对于内存访问之间使用同步来建模。如果你有一大堆需要并发建模的东西，而你的计算机又不能处理那么多的线程，就需要创建一个线程池并将你的操作在线程池中复用。

Go 语言在这个链条中加入了新的一环：*goroutine*。另外，Go 语言从著名的计算机科学家 Tony Hoare 那里借用了不少概念，并且给我们提供了新的原语来使用，即 *channel*。

如果继续按照线索推理，假设在操作系统的线程下提供新的抽象层会带来更多的复杂性，但有趣的是并没有。事实上，channel 使事情变得简单化。这是因为我们并没有真的在操作系统的线程之上增加了另一层的抽象层，我们取代了这些事物。

当然，线程依旧存在，但是我们发现几乎不在操作系统的线程层面考虑我们的问题了。取而代之的是，我们对于事物站在 goroutine 以及 channel 的角度

来进行思考，偶尔站在共享内存的角度来思考。这就引出了一些在本章后面的“如何帮助你”中探讨的有趣的特性。但是首先让我们更进一步看一下 Go 语言在哪里获得这些想法——Go 语言并发原语的根基论文：Tony Hoare 开创性的论文“Communicating Sequential Processes”。

什么是 CSP

当进行和 Go 语言有关讨论的时候，你会经常听到人们抛出 CSP 这个缩写。在某些环境下 CSP 经常被赞美成 Go 语言成功的原因以及并发编程的“万能钥匙”。它让不知道 CSP 的人开始认为计算机科学已经发现了一些可以像变魔术一样的方法让编写一个并发程序像编写一个串行程序一样简单。虽然 CSP 确实使这些变得更加简单，让程序变得更加健壮，但不幸的是它并不是一个奇迹。所以，CSP 到底是什么？为什么把大家都弄的如此兴奋？

CSP 即“Communicating Sequential Processes”（通信顺序进程），既是一个技术名词，也是介绍这种技术的论文的名字。在 1978 年，Charles Antony Richard Hoare 在 Association for Computing Machinery（一般被称作 ACM）中发表的论文。

在这篇论文里，Hoare 认为输入与输出是两个被忽略的编程原语，尤其是在并发代码中。在 Hoare 写作这篇论文的同时，关于如何架构程序的相关研究还在进行中，但是大部分的研究都是针对编写顺序代码的方法：`goto` 语句的使用正在被讨论，面向对象范型正在成为编程的基石。并发操作并没有被给予过多的思考。Hoare 开始纠正这个现象，所以，关于 CSP 的这篇论文就横空出世了。

在 1978 年的论文中，CSP 仅是一个完全用来展示通信顺序进程的能力的一个简单的编程语言。事实上，他甚至在论文中写道：

> 因此，本文介绍的概念和符号应该……不被认为适合作为一种编程语言，无论是抽象的还是具体的编程。

Hoare 深深的忧虑所展示的技术对未来的关于并发编程的研究没有任何作用，这种技术也许没有语言会真的按照他的想法来实现。在接下来的 6 年里，关于 CSP 的想法被提炼成了一个叫做“进程微积分”的正式名称来将 CSP 的想法投入到并发编程的实践。进程微积分是一种对并发系统进行数学化建模的方式，并且提供了代数法则来进行这些系统的变换来分析它们不同的属性，例如：并发与效率。尽管进程微积分是一个很有趣的主题，但是超出了本书的范围。而且正是因为 CSP 的原始论文以及从论文中进化而来的原语正是 Go 语言并发模型的主要灵感，而这正是我们接下来所要聚焦的。

用来支撑他关于输入与输出需要被按照语言的原语来考虑，Hoare 的 CSP 编程语言包含用来建模输入与输出，或者说“在进程间正确通信”（这就是论文名字的由来）的原语。Hoare 将进程这个术语运用到任何包含需要输入来运行且产生其他进程将会消费的输出的逻辑片段。Hoare 可能应该使用“函数”这个词汇，而不是在他写论文时在社区中关于如何构建程序的辩论。

为了在进程之间进行通信，Hoare 创造了输入与输出的命令：！代表发送输入到一个进程，？代表读取一个进程的输出。每一个指令都需要指定具体是一个输出变量（从一个进程中读取一个变量的情况），还是一个目的地（将输入发送到一个进程的情况）。有时，这两种方法会引用相同的东西，在这种情况下，这两个过程会被认为是相对应的。换言之，一个进程的输出应该直接流向另一个进程的输入。表 2-1 给我们展示了论文中的部分示例。

表 2-1：Hoare 的 CSP 论文中一些例子的摘录

操作	解释
`cardreader?card image`	从 `cardreader` 中读取一条记录，并将它的值（一个数组）赋给变量 `cardimage`
`lineprinter!line image`	向 `lineprinter` 发送 `lineimage` 的值
`x?(x,y)`	从名为 X 的进程输入一对值，并将它们分配给 x 和 y
`DIV!(3*a+b,13)`	处理 DIV，输出两个指定的值
`*[c:character; west?c → east!c]`	读取所有 west 输出的字符，然后输出到 east。当过程 west 结束时，终止

示例与 Go 语言的 channel 的相似性是显而易见的。注意最后一个示例中 `west` 的输出被送到一个变量 `c`，然后 `east` 的输入也是从相同的变量中接收的。这两个过程是相同的。在 Hoare 关于 CSP 的第一篇论文中，进程只能通过命名的源与目的进行通信。他承认这会让代码像一个函数库一样被嵌入到逻辑中，因为这段代码的消费者必须知道输入与输出的命名。他随意地提出将输入输出的名字注册成他称作“端口名”的可能性，也就是说，在并行命令的头部，需要声明名字，也就是我们大概可以认为是命名的参数与命名的返回值。

这种语言同时利用了一个所谓的守护命令，也就是 Edgar Dijkstra 在一篇之前在 1974 年所写的论文中介绍的，“Guarded commands, nondeterminacy and formal derivation of programs”。一个有守护的命令仅仅是一个带有左和右倾向的语句，由→来分割。左侧服务是有运行条件的，或者是守护右侧服务，如果左侧服务运行失败，或者在一个命令执行后，返回 false 或者退出，右侧服务永远不会被执行。将这些与 Hoare 的 I/O 命令组合起来，为 Hoare 的通信过程奠定了基础，从而实现了 channel。

使用这些原语，Hoare 运行了几个示例，并演示了如何以最佳的方式支持建模通信，从而使解决问题变得更简单、更容易理解。他使用的一些符号是简短的（Perl 程序员可能不同意），并且提出的问题有非常清晰的解决方案。类似的解决方案比较长一些，但也很清晰。

经验判断 Hoare 的建议是正确的，然而，有趣的是，在 Go 语言发布之前，很少有语言能够真正地为这些原语提供支持。大多数流行的语言都支持共享和内存访问同步到 CSP 的消息传递样式。当然也有例外，但不幸的是，这些都局限于没有广泛采用的语言。Go 语言是最早将 CSP 的原则纳入其核心的语言之一，并将这种并发编程风格引入到大众中。它的成功也使得其他语言尝试添加这些原语。

内存访问同步并不是天生就不好。我们将在本章后面“Go 语言的并发哲学”介绍，在 Go 语言中，甚至有时共享内存在某些情况下是合适的。但是，共享内存模型很难正确地使用，特别是在大型或复杂的程序中。正是由于这个原因，

并发被认为是 Go 语言的优势之一，它从一开始就建立在 CSP 的原则之上，因此很容易阅读、编写和推理。

如何帮助你

你或许不认为上述的内容都很迷人，但是，当你在读这本书的时候，你有可能有问题需要解决，以及在好奇为什么这些很重要，为什么 Go 语言相比于其他热门的语言，在并发上这么多的不同？

就像我们在本章前面的“并发与并行的区别”中讨论的对并发问题进行建模一样。通常来说，一种语言会将它们的抽象链结束在系统线程和内存访问同步的层级。Go 语言采用了一个不同的路线，并使用 goroutine 和 channel 来代替这些概念。

如果要画一个关于这两种并发代码的抽象画，我们很可能将 goruntine 类比成线程，把一个 channel 类比成一个 mutex（这些原语只是有相似之处，但愿这些对比可以帮助你找对方向）。这些不同的抽象可以为我们做什么呢？

goroutine 把我们从必须按照并行的方式思考中解放出来，作为替代，它允许我们按照更为自然的等级对问题进行建模。虽然我们已经复习了关于并发与并行的区别，而这些区别如何影响我们建模解决方案可能并不明确，让我们来开始一个示例。

比如说我需要构建一个端上请求字段的 Web 服务器。暂时先不用考虑架构，在一个仅提供线程抽象的语言中，很有可能思考下面的问题：

- 我的语言原生支持线程么，还是我需要选择一个类库？
- 我的进程限制边界应该是什么？
- 线程在操作系统中有多重量级？
- 我的程序需要运行的操作系统处理这些线程的时候有什么不同？
- 我需要创建一个工作线程池来承载创建的线程，该如何找到最佳的线程池大小？

这些都是需要考虑的很重要的事情，但是没有一个直接关系到你要解决的问题。你马上就被灌输如何解决并行问题的技术细节。

如果后退一步然后思考一下普通的问题，我们能做出如下陈述：独立的用户们链接到我的服务端并且打开一个会话。这个会话应该处理用户的请求并提供一个返回值。在 Go 语言中，可以立刻把这个问题的描述编写成代码：我们需要为每一个接入的链接创建一个 goroutine，在 goroutine 中处理请求（很有可能需要和其他的 goroutine 或者数据 / 服务进行交互），然后从 goroutine 的函数体中返回。我们对问题进行思考自然地直接映射为如何使用 Go 语言进行编码。

这是由一个 Go 语言给我们的保证所达成的：goroutine 是很轻量级的，我们通常情况下并不需要为创建新的 goroutine 的代价而担心。会有合适的机会让你去思考系统中有多少运行中的 goroutine，但是过早考虑的话，则是完完全全地过早优化。把这个和线程对比一下，你会发现提前考虑这些事情很明智。

某个语言有一个可以把并行抽象出来的框架，但并不意味着这种自然的方式对并发问题建模不重要！总有人必须去写这些框架，而你的代码就需要编写在开发者所构建的复杂的基础之上。复杂性在你编写代码的时候被隐藏了，并不意味着复杂性本身不存在，而复杂性正是滋生 bug 的温床。在 Go 语言下，这个语言就是围绕着并发进行设计的，所以说 Go 语言与它所提供的并发原语并无不一致的情况。这就意味着更少的阻力和更少的 bug！

一个更加自然的对于问题空间的映射有非常巨大的好处，不仅如此，它还有一些其他的有益的副作用。Go 语言的运行时自动地将 goroutine 映射到系统的线程上，并为我们管理它们之间的调度。这也就意味着对于运行时的优化可以在不改动我们如何对问题建模方式的情况下进行。随着并行技术的发展，Go 语言的运行时也会改进，程序的性能也会进步，所有的这些都是自然而然的。留意 Go 语言的发布版本通知，然后你有时会看到像这样的事情：

> 在 Go 1.5 中，goroutine 的调度顺序发生了改变。

Go 语言的开发者在幕后进行改进来使你的程序运行得更快。

并发与并行的解耦还有另一个好处：因为Go语言的运行时为你管理goroutine映射的调度，它可以在像goroutine阻塞等待I/O之类的事情上进行内省，从而智能地把OS的线程重新分配给没有被阻塞的goroutine。这也提高了你的代码性能。我们将在第6章讨论更多有关Go语言运行时的问题。

问题空间与Go语言代码之间的自然映射带来的另一个好处就是将问题空间建模为并发方式的数量增加了。因为我们作为开发者去解决问题，经常自然而然地按照并发的方式去处理。相比于我们可能使用的其他语言，在使用Go语言的时候我们会在更细的颗粒度级别编写并发代码。例如，如果回到网络服务器的例子上，现在在处理用户请求的时候都有一个独立的goroutine，而不是将链接绑定到一个线程池。更精细的粒度级别使我们的程序可以在运行到主机可能承载的并行数量的时候，可以动态地缩放，Amdahl法则的实践！这是非常惊人的。

goroutine仅仅是这个拼图的一部分。而其他来自CSP的概念，channel与select语句也增加了它的价值。

比如说，channel可以天然地和其他channel进行组合。这就使得编写大规模系统变得更加简单。因为你可以通过轻松地组合输出来协调多个子系统的输入。你可以将输入的channel与超时、取消或者消息组合到其他的子系统。而协调互斥体则是一个更加艰难的命题。

Go语言的`select`语句是对channel的一个补充，并且使多个通道组合的所有难点得以实现。`select`语句使你可以高效的等待事件，从一个竞争的channel中均匀、随机地选择一个消息，并在没有消息的时候继续等待等。

这个由CSP以及支撑运行时所启发的“漂亮挂毯”就是驱动Go语言的动力所在。我们将在这本书的剩余部分来探索这些东西是如何运转的，以及如何和为何使用它们来编写令人震惊的代码。

Go 语言的并发哲学

CSP 一直都是 Go 语言设计的重要组成部分。然而，Go 语言还支持通过内存访问同步和遵循该技术的原语来编写并发代码的传统方式。sync 与其他包中的结构体与方法可以让你执行锁，创建资源池取代 goroutine 等。

能够在 CSP 原语和内存访问同步之间选择对于你来说很棒，因为它让你去编写解决问题的并发代码上有了更多选择，但这可能显得有些莫名其妙。Go 语言的初学者总是认为 CSP 样式编写并发代码是 Go 语言编写并发代码的唯一方式。比如说，在 sync 包的文档中，有如下描述：

> sync 包提供了基本的同步基元，如互斥锁。除了 Once 类型和 WaitGroup 类型，大部分都是适用低水平程序线程，高水平的同步使用 channel 通信更好一些。

在 Go 语言的 FAQ 中，有如下陈述：

> 为了尊重 mutex，sync 包实现了 mutex，但是我们希望 Go 语言的编程风格将会激励人们尝试更高等级的技巧。尤其是考虑构建你的程序，以便一次只有一个 goroutine 负责某个特定的数据。
>
> 不要通过共享内存进行通信。相反，通过通信来共享内存。有数不清的关于 Go 语言核心团队的文章、讲座和访谈，相对于使用像 sync.Mutex 这样的原语，他们更加拥护 CSP。

因此，Go 语言团队为什么选择公开内存访问同步原语会感到困惑是完全可以理解的。更令人困惑的是，你通常会在外面看到出现的同步原语。见到人们抱怨过度使用 channel，也会听到一些 Go 语言团队成员说使用它们是“OK”的。Go 语言的维基上有一个关于此的引用：

> Go 语言的一个座右铭是，“使用通信来共享内存，而不是通过共享内存来通信。”

这就是说，Go 语言确实在 sync 包中提供了传统的锁机制。大多数的锁问题都可以通过 channel 或者传统的锁两者之一来解决。

所以说，我该用哪个？

使用最好描述和最简单的那个方式。

这是很好的建议，也是你在使用 Go 语言时经常看到的准则，但它有点含糊。我们如何理解什么更具表现力、更简单？我们应该使用什么标准？幸运的是，我们可以使用一些标准来帮助我们做正确的事情。正如我们将看到的那样，我们主要的区分方式来自于试图管理并发的地方：主观地想象一个狭窄的范围，或者在我们的系统外部。图 2-1 展示了这些用来创建决策树的准则。

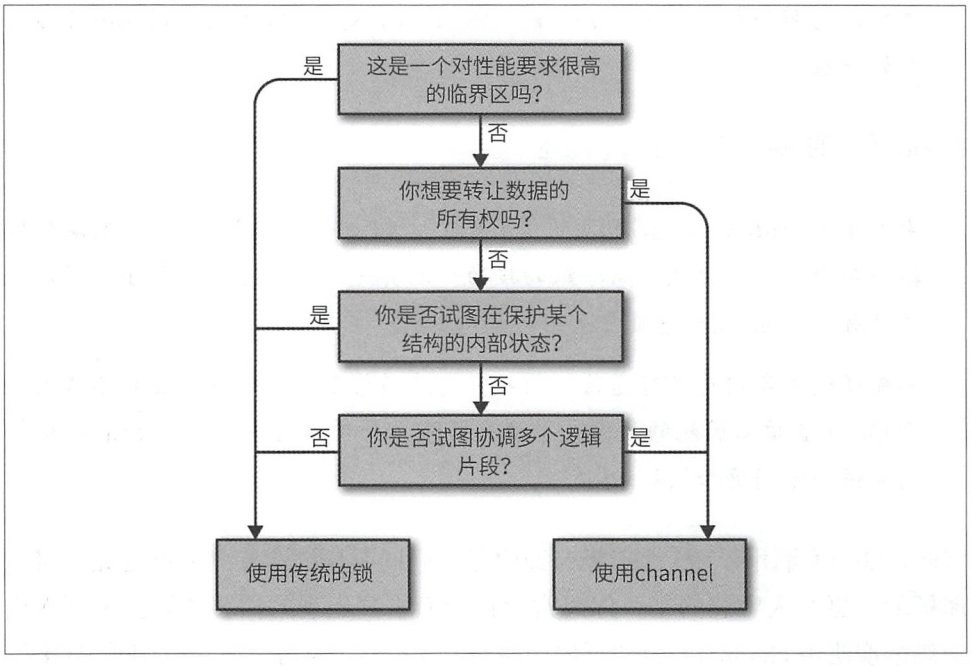


图 2-1：决策树

让我们逐步来介绍这些决策：

你想要转让数据的所有权么？

如果你有一块产生计算结果并想共享这个结果给其他代码块的代码，你所实际做的事情是传递了数据的所有权。如果你对内存所有制且不支持

GC 的语言很熟悉的话，对于这个概念你应该是很熟悉的：数据拥有所有者，并发程序安全就是保证同时只有一个并发上下文拥有数据的所有权。channel 通过将这个意图编写进 channel 类型本身来帮助我们表达这个意图。

这么做的一个很大的好处就是可以创建一个带缓存的 channel 来实现一个低成本的在内存中的队列来解耦你的生产者与消费者；另一个好处就是通过使用 channel 确保你的并发代码可以和其他的并发代码进行组合。

你是否试图在保护某个结构的内部状态？

这时候内存访问同步原语的一个很好的选择，也是一个你不应该使用 channel 的很好的示例。通过使用内存访问同步原语，可以为你的调用者隐藏关于重要代码块的实现细节。这是一个线程安全类型的小例子，且不会给调用者带来复杂性：

```
type Counter struct {
    mu sync.Mutex
    value int
}
func (c *Counter) Increment() {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.value++
}
```

如果你能回想起关于原子性的细节，可以说我们在这里所做的就是定义了 `Counter` 类型的原子性范围。调用增量可被认为是原子的。

记住这里的关键词是“内部的”。如果你发现自己正在将锁暴露在一个类型之外，这时候你应该注意了。试着将你的锁放在一个小的字典范围内。

你是否试图协调多个逻辑片段？

请记住，channel 本质上比内存访问同步原语更具可组合性。将锁分散在整个对象图中听起来像是一场噩梦，但是，将 channel 编写的随处可见是被鼓励以及期待的！我可以组合 channel，但是我不能轻易的组合锁或者有返回值的方法。

你会发现，因为 Go 语言的 `select` 语句，以及 channel 可以当作队列使用和被安全的随意传递。所以，当在使用 channel 的时候，你可以更简单的控制你软件中出现的激增的复杂性。如果你发现正在挣扎着理解你的并发

代码是如何工作的，为什么会出现死锁以及竞争，而你正在只用原语，这是一个你应该切换到 channel 的好示例。

这是一个对性能要求很高的临界区吗？

这绝对不意味着“我想让我的程序拥有高性能，因此，我应该只是用 mutex”。当然，如果你程序中的某部分，事实证明是一个主要的性能瓶颈，比程序的其他部分慢几个数量级，使用内存访问同步原语可能会帮助这个重要的部分在负载下执行。这是因为 channel 使用内存访问同步来操作，因此它们只能更慢。然而，在我们考虑这一点之前，性能至关重要的程序部分可能暗示着需要重新规划我们的程序。

希望这可以清楚地说明是否利用 CSP 风格的并发或内存访问同步。还有其他一些模式和做法在使用操作系统线程作为并发抽象方式的语言中很有用。在使用操作系统线程作为主要并发抽象的语言中还有其他的方式以及实践。比如说，像是线程池之类的东西经常出现。因为这些抽象大多数都是为了利用操作系统线程的优点与缺点。

Go 语言的并发性哲学可以这样总结：追求简洁，尽量使用 channel，并且认为 goroutine 的使用是没有成本的。

第 3 章

Go 语言并发组件

在本章中，我们将讨论 Go 语言的丰富特性，以及它如何支持并发。在本章结束时，你应该对语法、函数和包，以及它们的功能有一个很好的理解。

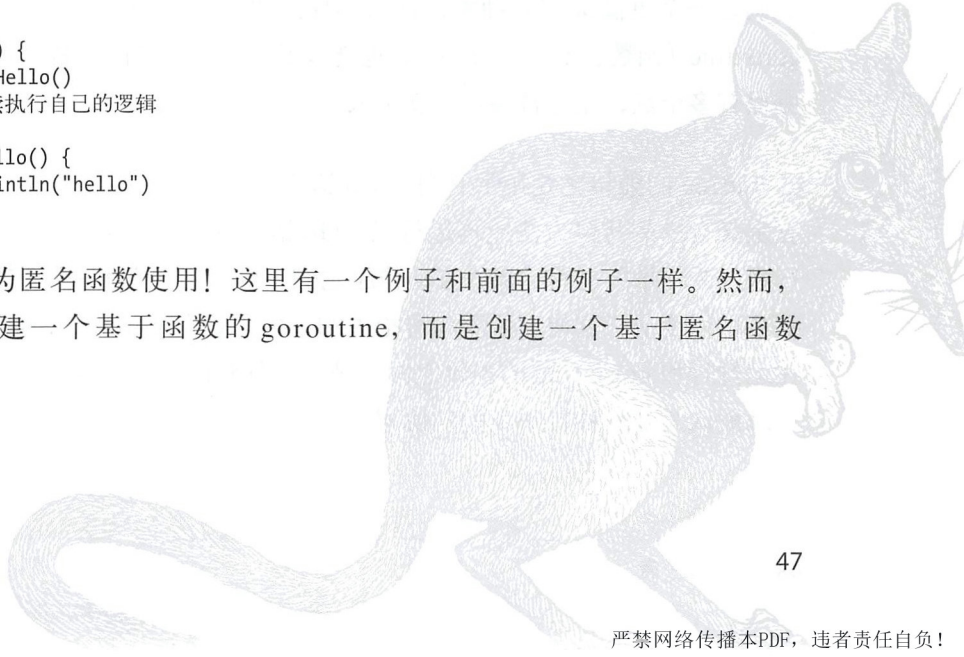
goroutine

goroutine 是 Go 语言程序中最基本的组织单位之一，所以我们要了解它们是什么以及如何工作。事实上每个 Go 语言程序都至少有一个 goroutine：*main goroutine*，它在进程开始时自动创建并启动。几乎在所有的项目中，你迟早会使用 goroutine 来解决 Go 语言编程遇到的问题。所以，它们是什么？

简单地说，goroutine 是一个并发的函数（记住：不一定是并行的），与其他代码一起运行。你可以简单地在一个函数之前添加 go 关键字来触发：

```
func main() {
    go sayHello()
    // 继续执行自己的逻辑
}
func sayHello() {
    fmt.Println("hello")
}
```

同样可以作为匿名函数使用！这里有一个例子和前面的例子一样。然而，我们不是创建一个基于函数的 goroutine，而是创建一个基于匿名函数 goroutine：



```
go func() {
    fmt.Println("hello")
}() ❶
// 继续执行自己的逻辑
```

❶ 请注意，我们必须立即在 go 关键字后面调用匿名函数来使用。

或者，你可以将该函数赋给一个变量，并将其命名为这样的匿名函数：

```
sayHello := func() {
    fmt.Println("hello")
}
go sayHello()
// 继续执行自己的逻辑
```

这个够酷！我们可以创建一个具有函数和单个关键字的并发逻辑块！信不信由你，你需要知道的就是创建 goroutine。关于如何正确地使用、同步，并组织它们，有很多需要说的，但了解这些知识需要你开始使用 goroutine。这一章的其余部分要深入分析 goroutine 是什么以及它们是如何工作的。如果你只对编写一些正确使用 goroutine 的代码感兴趣，你可以考虑跳到下一节。

那么让我们来看看这里发生了什么，goroutine 是如何工作的？它们是 OS 线程吗？绿色线程？我们能创造多少个 goroutine？

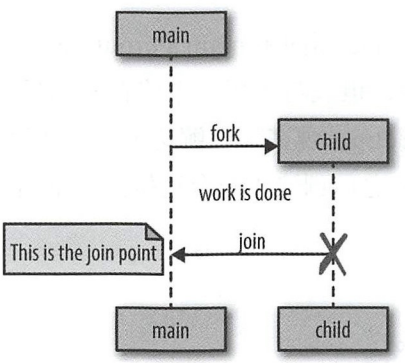
Go 语言中的 goroutine 是独一无二的（尽管其他的一些语言有类似的并发原语）。它们不是 OS 线程，也不是绿色线程（由语言运行时管理的线程），它们是一个更高级别的抽象，称为协程。协程是一种非抢占式的简单并发子 goroutine（函数、闭包或方法），也就是说，它们不能被中断。取而代之的是，协程有多个点，允许暂停或重新进入。

goroutine 的独特之处在于它们与 Go 语言的运行时的深度集成。goroutine 没有定义自己的暂停方法或再运行点。Go 语言的运行时会观察 goroutine 的运行时行为，并在它们阻塞时自动挂起它们，然后在它们不被阻塞时恢复它们。在某种程度上，这使它们成为可抢占的，但只是在 goroutine 被阻塞的情况。在运行时和 goroutine 的逻辑之间，是一种优雅的伙伴关系。因此，goroutine 可以被认为是一种特殊类型的协程。

协程和 goroutine 都是隐式并发结构，但并发并不是协程的属性：必须同时托管多个协程，并给每个协程一个执行的机会。否则，它们就不会并发！请注意，这并不意味着协程是隐式并行的。当然有可能有几个协程按顺序并行执行的假象，事实上，这种情况一直在发生。

Go 语言的主机托管机制是一个名为 *M:N* 调度器的实现，这意味着它将 *M* 个绿色线程映射到 *N* 个 OS 线程。然后将 goroutine 安排在绿色线程上。当我们的 goroutine 数量超过可用的绿色线程时，调度程序将处理分布在可用线程上的 goroutine，并确保当这些 goroutine 被阻塞时，其他的 goroutine 可以运行。我们将在第 6 章讨论这些细节，但是在这里我们将介绍 Go 语言的并发模型。

Go 语言遵循一个称为 *fork-join* 的并发模型[注 1]。fork 这个词指的是在程序中的任意一点，它可以将执行的子分支与其父节点同时运行。*join* 这个词指的是，在将来某个时候，这些并发的执行分支将会合并在一起。这里有一个示意图，帮助你描绘它：



Go 语言是如何执行 fork 的，执行的子线程是 goroutine。让我们回到简单的 goroutine 例子：

```
sayHello := func() {
    fmt.Println("hello")
}
go sayHello()
// 继续执行自己的逻辑
```

注 1：熟悉 C 的人可能正在将这个模型和 `fork` 函数进行比较。fork-join 模型是控制执行并发性的逻辑模型。在逻辑级别上，它确实描述了一个调用 fork 的 C 程序，然后 `wait`。fork-join 模型没有提到内存是如何管理的。

在这里，sayHello 函数将在 goroutine 上运行，而程序的其余部分将继续执行。在本例中，没有 join 点。执行 sayHello 的 goroutine 将在未来的某个不确定的时间退出，而程序的其余部分将会继续执行。

但是，这个例子有一个问题：正如上面所写的程序，它不确定 sayHello 函数是否会运行。goroutine 将会被创建，并计划在 Go 语言运行时执行，但是它实际上可能没有机会在 main goroutine 退出之前运行。

实际上，因为我们省略了 main 函数的其余部分，为了简单起见，当运行这个小示例时，几乎可以肯定的是，程序将在 goroutine 被系统调用之前完成执行。因此，你不会看到“hello”这个词被打印到 stdout。你可以在创建 goroutine 之后执行 time.Sleep，但是要记住，这实际上并没有创建一个 join 点，只有一个竞争条件。如果回顾第 1 章，你增加了 goroutine 在程序退出前执行的概率，但你并不能保证一定会执行。join 点是保证程序正确性和消除竞争条件的关键。

为了创建一个 join 点，你必须同步 main goroutine 和 sayHello goroutine。这可以通过多种方式实现，但我将使用一个将在本章后面的“sync 包”中讨论的内容：sync.Waitgroup。现在，理解这个示例如何创建连接点并不重要，它只是在两个 goroutine 之间创建了一个连接点。下面是一个正确的例子：

```
var wg sync.WaitGroup
sayHello := func() {
    defer wg.Done()
    fmt.Println("hello")
}
wg.Add(1)
go sayHello()
wg.Wait()❶
```

❶ 这就是连接点的使用方式。

输出如下：

```
hello
```

这个例子将决定 main goroutine，直到 goroutine 托管 sayHello 函数为止。你将在本章后面的“sync 包”中了解 sync.WaitGroup 的工作原理，但是为了使我们的示例正常运行，我将开始使用它创建 join 点。

我们在示例中使用了许多匿名函数来创建快速 goroutine 样例。让我们把注意力转移到闭包上。闭包可以从创建它们的作用域中获取变量。如果你在 goroutine 中运行一个闭包，那么闭包是在这些变量的副本上运行，还是原值的引用上运行？让我们试试看：

```
var wg sync.WaitGroup
salutation := "hello"
wg.Add(1)
go func() {
    defer wg.Done()
    salutation = "welcome"❶
}()
wg.Wait()
fmt.Println(salutation)
```

❶ 在这里，我们看到 goroutine 修改了变量的 salutation 值。

你认为变量的 salutation 值是什么？“hello”还是“welcome”？让我们运行它并把它找出来：

```
welcome
```

有意思！事实证明，goroutine 在它们所创建的相同地址空间内执行，因此我们的程序打印出“welcome”这个词。让我们再看一个例子。你认为这个程序会输出什么？

```
var wg sync.WaitGroup
for _, salutation := range []string{"hello", "greetings", "good day"} {
    wg.Add(1)
    go func() {
        defer wg.Done()
        fmt.Println(salutation) ❶
    }()
}
wg.Wait()
```

❶ 在这里，我们引用了字符串类型的切片作为创建循环变量的 salutation 值。

答案比大多数人想象的要复杂得多，而且是为数不多的令人惊讶的事情之一。大多数人直觉上认为这将会不确定顺序地打印出“hello”“greetings”和“good day”，但看看它做了什么：

```
good day
good day
good day
```

令人意想不到！我们来看看这里发生了什么。在这个示例中，goroutine 正在运行一个闭包，该闭包使用变量 `salutation` 时，字符串的迭代已经结束。当我们循环迭代时，`salutation` 被分配到 slice literal 中的下一个字符串值。因为计划中的 goroutine 可能在未来的任何时间点运行，它不确定在 goroutine 中会打印出什么值。在我的机器上，在 goroutine 开始之前循环有很高的概率会退出。这意味着变量的 `salutation` 值不在范围之内。然后会发生什么呢？goroutine 还能引用一些已经超出范围的东西吗？ goroutine 不会访问那些可能被垃圾回收的内存吗？

这是一个关于如何管理内存的有趣的点。Go 语言运行时会足够小心地将对变量 `salutation` 值的引用仍然保留，由内存转移到堆，以便 goroutine 可以继续访问它。

通常在我的计算机上，在任何 goroutine 开始运行之前，循环就会退出，所以 `salutation` 会被转移到堆中，在我的字符串切片中引用最后一个值“good day”。所以我通常会看到三次“good day”。编写这个循环的正确方法是将 `salutation` 的副本传递到闭包中，这样当 goroutine 运行时，它将从循环的迭代中操作数据：

```
var wg sync.WaitGroup
for _, salutation := range []string{"hello", "greetings", "good day"} {
    wg.Add(1)
    go func(salutation string) {❶
        defer wg.Done()
        fmt.Println(salutation)
    }(salutation)❷
```

```
    }
    wg.Wait()
```

❶ 声明一个参数，就像其他函数一样。把原来变量的 `salutation` 显式的映射到闭包中。

❷ 我们将当前迭代的变量传递给闭包。创建了一个字符串结构的副本，从而确保当 goroutine 运行时，我们可以引用适当的字符串。

如你所见，我们得到了正确的输出：

```
good day
hello
greetings
```

这个示例的行为与我们期望的一样，只是稍微有点烦琐。

因为这些 goroutine 在相同的地址空间中运行，并且只有简单的宿主函数，所有使用 goroutine 编写非并发代码是非常自然的。Go 语言的编译器很好地处理了内存中的变量，这样 goroutine 就不会意外地访问被释放的内存，这使得开发人员可以专注于他们的问题空间而不是内存管理。然而，这不是一张空白支票。

由于多个 goroutine 可以在同一个地址空间上运行，所以我们仍然需要担心同步问题。正如我们已经讨论过的，我们可以选择同步访问 goroutine 访问的共享内存，或者可以使用 CSP 原语通过通信来共享内存。我们稍后将在本章后面的“Channel”和“sync 包”中讨论这些技术。

goroutine 的另一个好处是它们非常轻。下面是“Go 语言 FAQ”的摘录：

> 一个新创建的 goroutine 被赋予了几千字节，这在大部分情况都是足够的。当它不运行时，Go 语言运行时就会自动增长（缩小）存储堆栈的内存，允许许多 goroutine 存在适当的内存中。每个函数调用 CPU 的开销平均为 3 个廉价指令。在同一个地址空间中创建成千上万的 goroutine 是可行的。如果 goroutine 只是线程，系统的资源消耗会更小。

每个 goroutine 几千字节，这并没有什么问题！让我们来验证一下。但是在我们开始之前，我们必须讨论一个关于 goroutine 有趣的事情：GC 并没有回收被丢弃的 goroutine。如果我写如下代码：

```
go func() {
    // 将永远阻塞的操作
}()
// 开始工作
```

这里的 goroutine 将一直存在直到进程退出。我们将在第 4 章的“防止 goroutine 泄漏”中讨论如何解决这一问题。在下一个例子中，我们将利用这一点来实际测算 goroutine 的大小。

在下面的例子中，我们将 goroutine 不被 GC 的事实与运行时的自省能力结合起来，并测算在 goroutine 创建之前和之后分配的内存数量：

```
memConsumed := func() uint64 {
    runtime.GC()
    var s runtime.MemStats
    runtime.ReadMemStats(&s)
    return s.Sys
}

var c <-chan interface{}
var wg sync.WaitGroup
noop := func() { wg.Done(); <-c } ❶

const numGoroutines = 1e4 ❷
wg.Add(numGoroutines)
before := memConsumed()  ❸
for i := numGoroutines; i > 0; i-- {
    go noop()
}
wg.Wait()
after := memConsumed() ❹
fmt.Printf("%.3fkb", float64(after-before)/numGoroutines/1000)
```

❶ 我们需要一个永远不会退出的 goroutine，这样就可以在内存中保留一段时间用于测算。不要担心我们是如何实现这个目标的，只要了解这个 goroutine 不会退出，直到进程结束。

❷ 定义了要创建的 goroutine 的数量。我们将用大数定律，渐渐地接近一个 goroutine 的大小。

❸ 测算在创建 goroutine 之前消耗的内存总量。

❹ 测算在创建 goroutine 之后消耗的内存总量。

结果如下：

```
2.817KB
```

看起来文档是正确的！这些都是空的 goroutine，什么都不做，但它仍然让我们知道可能创造的 goroutine 的数量。表 3-1 给出了一些粗略的估计，在不使用交换空间的情况下你可以使用 64 位 CPU 创建多少 goroutine。

表 3-1：在给定的内存中，对可能出现的 goroutine 的数量进行分析

内存 (GB)	goroutines(#/100000)	数量级
2^0	3.718	3
2^1	7.436	3
2^2	14.873	6
2^3	29.746	6
2^4	59.492	6
2^5	118.983	6
2^6	237.967	6
2^7	475.934	6
2^8	951.867	6
2^9	1903.735	9

这些数字相当大！在我的笔记本计算机上有 8GB 的内存，这意味着理论上可以在不使用交换空间的情况下启动数百万的 goroutine。当然，忽略了在我的计算机上运行的其他东西，以及 goroutine 的实际内容，但是这个快速的计算表明了 goroutine 是多么的轻量级！

可能会影响性能的是上下文切换，即当一个被托管的并发进程必须保存它的状态以切换到一个不同的运行并发进程时。如果我们有太多的并发进程，可能会将所有的 CPU 时间消耗在它们之间的上下文切换上，而没有资源完成任何真正需要 CPU 的工作。在操作系统级别，使用线程可能非常昂贵。OS 线

程必须保存如寄存器值、查找表和内存映射之类的东西，以便能够在有限的时间内成功地切换回当前线程。然后，它必须为传入的线程加载相同的信息。

软件中的上下文切换相对来说要廉价得多。在一个软件定义的调度器下，运行时可以更有选择性地保存数据用于检索，如何持久化，以及何时需要持久化。让我们来看看在 OS 线程和 goroutine 之间切换的上下文的相对性能。首先，我们将利用 Linux 的内置基准测试套件来度量在相同核心的两个线程之间发送消息需要多长时间：

```
taskset -c 0 perf bench sched pipe -T
```

输出如下：

```
# Running 'sched/pipe' benchmark:
# Executed 1000000 pipe operations between two threads

     Total time: 2.935 [sec]

       2.935784 usecs/op
         340624 ops/sec
```

这个基准实际上度量了在线程上发送和接收消息所需的时间，因此我们将计算结果并将其除以 2。我们用了 1.467μs 来进行上下文切换。这看起来不算太糟，但还是保留判断，直到我们检查 goroutine 之间的上下文切换。

我们将使用 Go 语言构建一个类似的基准。我已经用到了一些我们还没有讨论过的东西，所以如果有什么让人困惑的东西，标注一下，把注意力集中在结果上。下面的示例将创建两个 goroutine 并在它们之间发送一条消息：

```
func BenchmarkContextSwitch(b *testing.B) {
    var wg sync.WaitGroup
    begin := make(chan struct{})
    c := make(chan struct{})

    var token struct{}
    sender := func() {
        defer wg.Done()
        <-begin ❶
        for i := 0; i < b.N; i++ {
```

```
            c <- token ❷
        }
    }
    receiver := func() {
        defer wg.Done()
        <-begin ❶
        for i := 0; i < b.N; i++ {
            <-c ❸
        }
    }

    wg.Add(2)
    go sender()
    go receiver()
    b.StartTimer() ❹
    close(begin) ❺
    wg.Wait()
}
```

❶ 我们在这里等待，直到被告知开始执行。我们对上下文切换度量的时候，不需要考虑设置和启动每个 goroutine 的成本。

❷ 我们将消息发送到接收器 goroutine。一个 `struct{}{}` 被称为一个空结构，它没有内存占用，因此，我们只是在发出信号的时候记录时间。

❸ 收到一条信息，但什么也不做。

❹ 开始计时。

❺ 告诉两个 goroutine 开始运行。

运行基准测试，假设我们只希望使用一个 CPU，以便它对 Linux 基准测试是一个类似的测试。让我们来看看结果：

```
    go test -bench=. -cpu=1 \
    src/gos-concurrency-building-blocks/goroutines/fig-ctx-switch_test.go

BenchmarkContextSwitch    5000000                225      ns/op
PASS
ok                        command-line-arguments    1.393s
```

每个上下文切换需要 225ns，哇！如果你还记得 1.467μs，0.225μs 或 92% 的速度提升比我的计算机一个操作系统上下文切换还要快。很难断言有多少

goroutine 会导致上下文切换过于频繁，但是可以很轻松地说，上限很可能不会成为使用 goroutine 的任何障碍。

读了这一节之后，你现在应该了解了如何开始使用 goroutine，并了解了它们是如何工作的。你也应该有信心，只要有需求场景，你就可以安全地创建一个 goroutine。正如我们在第 2 章的“并发与并行的差异”中所讨论的那样，你创建的 goroutine 越多，如果你的问题空间中每个并发代码段都不受任一 Amdahl 定律的约束，那么你的程序就会使用多个处理器进行扩展。创建 goroutine 非常廉价，只有当你已经证明了它们是性能问题的根本原因后，你才应该讨论它们的成本。

sync 包

sync 包包含对低级别内存访问同步最有用的并发原语。如果你使用的语言主要通过内存访问同步来处理并发，那么你可能已经熟悉了这些类型。Go 语言和这些语言之间的区别在于，Go 语言已经在内存访问同步原语之上构建了一组新的并发原语，以向你提供一组扩展的工作。正如我们在第 2 章“Go 语言的并发哲学”中所讨论的，这些操作都有它们的用途，主要是在诸如 `struct` 这样的小范围内。由你决定何时进行内存访问同步。说到这里，让我们开始看一下 sync 包公开的各种原语。

WaitGroup

当你不关心并发操作的结果，或者你有其他方法来收集它们的结果时，`WaitGroup` 是等待一组并发操作完成的好方法。如果这两个条件都不满足，我建议你使用 channel 和 `select` 语句。`WaitGroup` 非常有用，我首先介绍它，以便在后续部分使用它。下面是一个使用 `WaitGroup` 等待 goroutine 完成的基本例子：

```
var wg sync.WaitGroup

wg.Add(1)      ❶
```

```
    go func() {
        defer wg.Done()    ❷
        fmt.Println("1st goroutine sleeping...")
        time.Sleep(1)
    }()

    wg.Add(1)    ❶
    go func() {
        defer wg.Done()    ❷
        fmt.Println("2nd goroutine sleeping...")
        time.Sleep(2)
    }()

    wg.Wait()    ❸
    fmt.Println("All goroutines complete.")
```

❶ 调用 Add，参数为 1，表示一个 goroutine 开始了。

❷ 使用 defer 关键字来确保在 goroutine 退出之前执行 Done 操作，我们向 WaitGroup 表明我们已经退出了。

❸ 执行 Wait 操作，这将阻塞 main goroutine，直到所有 goroutine 表明它们已经退出。

输出如下：

```
2nd goroutine sleeping...
1st goroutine sleeping...
All goroutines complete.
```

你可以将 WaitGroup 视为一个并发－安全的计数器：调用通过传入的整数执行 add 方法增加计数器的增量，并调用 Done 方法对计数器进行递减。Wait 阻塞，直到计数器为零。

注意，添加的调用是在他们帮助跟踪的 goroutine 之外完成的。如果我们不这样做，我们就会引入一种竞争条件，因为在本章前面“goroutines”中，我们不能保证 goroutine 何时会被调度，可以在 goroutine 开始调度前调用 Wait 方法。如果将调用 Add 的方法添加到 goroutine 的闭包中，那么 Wait 调用可能会直接返回，而且不会阻塞，因为 Add 调用不会发生。

通常情况下，都要尽可能地向它们正在帮助追踪的 goroutine 中添加尽可能多的信息，但有时你会发现只调用一次 Add 来追踪一组 goroutine。我通常在这样的循环之前执行这种操作：

```
hello := func(wg *sync.WaitGroup, id int) {
    defer wg.Done()
    fmt.Printf("Hello from %v!\n", id)
}

const numGreeters = 5
var wg sync.WaitGroup
wg.Add(numGreeters)
for i := 0; i < numGreeters; i++ {
    go hello(&wg, i+1)
}
wg.Wait()
```

输出如下：

```
Hello from 5!
Hello from 4!
Hello from 3!
Hello from 2!
Hello from 1!
```

互斥锁和读写锁

如果你已经熟悉通过内存访问同步处理并发的语言，那么你可能会立即识别 Mutex 互斥锁。如果你不把自己算在内，别担心，Mutex 很容易理解。Mutex 是“互斥”的意思，是保护程序中临界区的一种方式。如果你还记得第 1 章，临界区是你程序中需要独占访问共享资源的区域。Mutex 提供了一种安全的方式来表示对这些共享资源的独占访问。为了使用一个资源，channel 通过通信共享内存，而 Mutex 通过开发人员的约定同步访问共享内存。你可以通过使用 Mutex 对内存进行保护来协调对内存的访问。这里有一个简单的例子，两个 goroutine 试图增加和减少一个共同的值，它们使用 Mutex 互斥锁来同步访问：

```
var count int
var lock sync.Mutex
```


```
increment := func() {
    lock.Lock()  ❶
    defer lock.Unlock()  ❷
    count++
    fmt.Printf("Incrementing: %d\n", count)
}

decrement := func() {
    lock.Lock()          ❶
    defer lock.Unlock()    ❷
    count--
    fmt.Printf("Decrementing: %d\n", count)
}

// 增量
var arithmetic sync.WaitGroup
for i := 0; i <= 5; i++ {
    arithmetic.Add(1)
    go func() {
        defer arithmetic.Done()
        increment()
    }()
}

// 减量
for i := 0; i <= 5; i++ {
    arithmetic.Add(1)
    go func() {
        defer arithmetic.Done()
        decrement()
    }()
}

arithmetic.Wait()
fmt.Println("Arithmetic complete.")
```

❶ 我们请求对临界区的独占（这个例子里的计数器）使用互斥锁来解决。

❷ 我们指出已经完成了对临界区锁定的保护。

输出如下：

```
Decrementing: -1
Incrementing: 0
Decrementing: -1
Incrementing: 0
Decrementing: -1
Decrementing: -2
Decrementing: -3
Incrementing: -2
Decrementing: -3
```


```
Incrementing: -2
Incrementing: -1
Incrementing: 0
Arithmetic complete.
```

你会注意到，我们总是在 defer 语句中调用 Unlock。这是一个十分常见的习惯用法，它使用 Mutex 互斥锁来确保即使出现了 panic，调用也总是发生。如果不这样做，可能会导致程序陷入死锁。

关键部分之所以如此命名，是因为它们反映了程序中的瓶颈。进入和退出一个临界区是有消耗的，所以一般人会尽量减少在临界区的时间。

这样做的一个策略是减少临界区的范围。可能存在需要在多个并发进程之间共享内存的情况，但可能这些进程不是都需要读写此内存。如果是这样，你可以利用不同类型的互斥对象：sync.RWMutex。

sync.RWMutex 在概念上和互斥是一样的：它守卫着对内存的访问，然而，RWMutex 让你对内存有了更多控制。你可以请求一个锁用于读处理，在这种情况下你将被授予访问权限，除非该锁被用于写处理。这意味着，任意数量的读消费者可以持有一个读锁，只要没有其他事物持有一个写锁。这里有一个例子，它演示了一个生产者，它不像代码中创建的众多消费者那样活跃：

```
producer := func(wg *sync.WaitGroup, l sync.Locker) {  ❶
    defer wg.Done()
    for i:=5;i>0;i-- {
        l.Lock()
        l.Unlock()
        time.Sleep(1)  ❷
    }
}

observer := func(wg *sync.WaitGroup, l sync.Locker) {
    defer wg.Done()
    l.Lock()
    defer l.Unlock()
}

test := func(count int, mutex, rwMutex sync.Locker) time.Duration {
    var wg sync.WaitGroup
    wg.Add(count+1)
    beginTestTime := time.Now()
    go producer(&wg, mutex)
```


```
    for i:=count;i>0;i-- {
        go observer(&wg, rwMutex)
    }

    wg.Wait()
    return time.Since(beginTestTime)
}

tw := tabwriter.NewWriter(os.Stdout, 0, 1, 2, ' ', 0)
defer tw.Flush()

var m sync.RWMutex
fmt.Fprintf(tw, "Readers\tRWMutext\tMutex\n")
for i:=0;i<20;i++ {
    count := int(math.Pow(2, float64(i)))
    fmt.Fprintf(
        tw,
        "%d\t%v\t%v\n",
        count,
        test(count, &m, m.RLocker()),
        test(count, &m, &m),
    )
}
```

❶ producer 函数的第二个参数是 sync.Locker 类型。这个接口有两个方法 Lock 和 Unlock，分别对应 Mutex 和 RWMutex 类型。

❷ 我们让 producer 等待 1s，使它比观察者的 goroutines 更不活跃。

输出如下：

```
Readers RWMutext      Mutex
1       38.343µs      15.854µs
2       21.86µs       13.2µs
4       31.01µs       31.358µs
8       63.835µs      24.584µs
16      52.451µs      78.153µs
32      75.569µs      69.492µs
64      141.708µs     163.43µs
128     176.35µs      157.143µs
256     234.808µs     237.182µs
512     262.186µs     434.625µs
1024    459.349µs     850.601µs
2048    840.753µs     1.663279ms
4096    1.683672ms    2.42148ms
8192    2.167814ms    4.13665ms
16384   4.973842ms    8.197173ms
32768   9.236067ms    16.247469ms
65536   16.767161ms   30.948295ms
```


```
131072  71.457282ms   62.203475ms
262144  158.76261ms   119.634601ms
524288  303.865661ms  231.072729ms
```

可以看到，在这个特殊的例子中减少了我们的临界区的横截面，实际上只给开始的 2^{13} 个读消费者的返回信息。这将取决于你的临界区在做什么，但是通常建议使用 RWMutex，而不是 Mutex，因为它在逻辑上更加合理。

cond

对于 cond 类型的注释确实很好地描述了它的用途：

> ……一个 goroutine 的集合点，等待或发布一个 event。

在这个定义中，一个“event”是两个或两个以上的 goroutine 之间的任意信号，除了它已经发生的事实外，没有任何信息。通常情况下，在 goroutine 继续执行之前，你需要等待其中一个信号。如果我们要研究如何在没有 Cond 类型的情况下实现这一目标，一个简单的方法就是使用无限循环：

```
for conditionTrue() == false {
}
```

然而，这将消耗一个 CPU 核心的所有周期。为了解决这个问题，我们可以引入一个 time.Sleep。

```
for conditionTrue() == false {
    time.Sleep(1*time.Millisecond)
}
```

这样更好，但它仍然是低效的，而且你必须弄清楚要等待多久：太长，会人为地降低性能；太短，会不必要地消耗太多的 CPU 时间。如果有一种方法可以让 goroutine 有效地等待，直到它发出信号并检查它的状态，那就更好了。这正是 Cond 类型为我们所做的。使用 Cond，我们可以这样编写前面例子的代码：

```
c := sync.NewCond(&sync.Mutex{}) ❶
c.L.Lock()❷
for conditionTrue() == false {
    c.Wait()❸
}
c.L.Unlock()  ❹
```

❶ 我们实例化一个新的 cond。NewCond 函数创建一个类型，满足 sync.Locker 接口。这使得 cond 类型能够以一种并发安全的方式与其他 goroutine 协调。

❷ 我们锁定了这个条件。这是必要的，因为在进入 Locker 的时候，执行 Wait 会自动执行 Unlock。

❸ 等待通知，条件已经发生。这是一个阻塞通信，goroutine 将被暂停。

❹ 我们为这个条件 Locker 执行解锁操作。这是必要的，因为当执行 Wait 退出操作的时候，它会在 Locker 上调用 Lock 方法。

这种方法效率更高。注意，调用 Wait 不只是阻塞，它挂起了当前的 goroutine，允许其他 goroutine 在 OS 线程上运行。当你调用 Wait 时，会发生一些其他事情：进入 Wait 后，在 Cond 变量的 Locker 上调用 Unlock 方法，在退出 Wait 时，在 Cond 变量的 Locker 上执行 Lock 方法。在我看来，这需要慢慢习惯，它实际上是方法的一个隐藏的副作用。看起来我们在等待条件发生的时候一直持有这个锁，但事实并非如此。当你浏览代码时，你需要留意这个模式。

让我们扩展这个例子，并显示等式的两边：等待信号的 goroutine 和发送信号的 goroutine。假设我们有一个固定长度为 2 的队列，还有 10 个我们想要推送到队列中的项目。我们想要在有房间的情况下尽快排队，所以就希望在队列中有空间时能立即得到通知。让我们尝试使用 Cond 来管理这种调度：

```
c := sync.NewCond(&sync.Mutex{})❶
queue := make([]interface{}, 0, 10) ❷

removeFromQueue := func(delay time.Duration) {
    time.Sleep(delay)
    c.L.Lock()  ❸
```


```
        queue = queue[1:]  ❾
        fmt.Println("Removed from queue")
        c.L.Unlock() ❿
        c.Signal() ⓫
    }

    for i:=0;i<10;i++{
        c.L.Lock()  ❸
        for len(queue) == 2 {  ❹
            c.Wait()  ❺
        }
        fmt.Println("Adding to queue")
        queue = append(queue, struct{}{})
        go removeFromQueue(1*time.Second)  ❻
        c.L.Unlock()  ❼
    }
```

❶ 首先，我们使用标准的 sync.Mutex 作为锁。

❷ 接下来，我们创建一个长度为 0 的切片。因为我们最终会添加 10 个项目，所以用 10 的容量实例化它。

❸ 我们通过在条件的锁存器上调用锁来进入临界区。

❹ 检查一个循环中队列的长度。这很重要，因为在这种情况下的信号并不一定意味着是你所等待的信号，也可能只是发生了什么。

❺ 调用 Wait，这将暂停 main goroutine 直到一个信号的条件已经发送。

❻ 创建了一个新的 goroutine，它将在一秒钟后删除一个元素。

❼ 退出条件的临界区，因为我们已经成功地进入了一个项目。

❽ 再次进入临界区，以便我们可以修改与条件相关的数据。

❾ 通过将切片的头部重新分配到第二个项目来模拟对一个项目的排队。

❿ 退出条件的临界区，因为我们已经成功地删除了一个项目。

⓫ 我们让一个正在等待的 goroutine 知道发生了什么事情。

输出如下：

```
Adding to queue
Adding to queue
Removed from queue
Adding to queue
```


```
Removed from queue
Adding to queue
Removed from queue
Adding to queue
Removed from queue
Adding to queue
Removed from queue
Adding to queue
Removed from queue
Adding to queue
Removed from queue
Adding to queue
Removed from queue
Adding to queue
```

如你所见，该程序成功地将所有 10 个项目添加到队列中（并且在它有机会将前两项删除之前退出）。它也总是等待，直到至少有一个项目被排入队列，然后再进行另一个项目。

在这个例子中，我们还有一个新方法，`Signal`。这是 `Cond` 类型提供的两种方法中的一种，它提供通知 goroutine 阻塞的调用 `Wait`，条件已经被触发。另一种方法叫做 `Broadcast`。运行时内部维护一个 FIFO 列表，等待接收信号；`Signal` 发现等待最长时间的 goroutine 并通知它，而 `Broadcast` 向所有等待的 goroutine 发送信号。`Broadcast` 可以说是这两种方法中比较有趣的一种，因为它提供了一种同时与多个 goroutine 通信的方法。我们可以通过 channel（在本章后面“Channels”中介绍）对信号进行简单的复制，但是重复调用 `Broadcast` 的行为将会更加困难。此外，与利用 channel 相比，`Cond` 类型的性能要高很多。

为了了解使用 `Broadcast` 的方法，让我们假设正在创建一个带有按钮的 GUI 应用程序。我们想注册任意数量的函数，当该按钮被单击时，它将运行。`Cond` 可以完美胜任，因为我们可以使用它的 `Broadcast` 方法通知所有注册的处理程序。让我们看看它的例子：

```
type Button struct {  ❶
    Clicked *sync.Cond
}
button := Button{ Clicked: sync.NewCond(&sync.Mutex{}) }

subscribe := func(c *sync.Cond, fn func()) { ❷
```


```
    var goroutineRunning sync.WaitGroup
    goroutineRunning.Add(1)
    go func() {
        goroutineRunning.Done()
        c.L.Lock()
        defer c.L.Unlock()
        c.Wait()
        fn()
    }()
    goroutineRunning.Wait()
}

var clickRegistered sync.WaitGroup  ❸
clickRegistered.Add(3)
subscribe(button.Clicked, func() {  ❹
    fmt.Println("Maximizing window.")
    clickRegistered.Done()
})
subscribe(button.Clicked, func() {  ❺
    fmt.Println("Displaying annoying dialog box!")
    clickRegistered.Done()
})
subscribe(button.Clicked, func() {  ❻
    fmt.Println("Mouse clicked.")
    clickRegistered.Done()
})

button.Clicked.Broadcast()   ❼

clickRegistered.Wait()
```

❶ 定义了一个 Button 类型，它包含一个条件，Clicked。

❷ 定义了一个便利构造函数，它允许我们注册函数处理来自条件的信号。每个处理程序都在自己的 goroutine 上运行，并且订阅不会退出，直到 goroutine 被确认运行为止。

❸ 我们为鼠标按键事件设置了一个处理程序。它反过来调用 Cond 上的 Broadcast，让所有的处理程序都知道鼠标按键已经被单击了（更健壮的实现将首先检查它是否已经被抑制）。

❹ 创建一个 WaitGroup。这只是为了确保我们的程序在写入 stdout 之前不会退出。

❺ 注册一个处理程序，当单击按键时，它将模拟最大化按钮的窗口。

❻ 注册一个处理程序，该处理程序在单击鼠标时模拟显示对话框。

❼ 接下来，我们模拟一个用户通过单击应用程序的按钮来单击鼠标按键。

输出如下：

```
Mouse clicked.
Maximizing window.
Displaying annoying dialog box!
```

你可以看到，在Clicked Cond上调用Broadcast，所有三个处理程序都将运行。如果不是clickRegistered的WaitGroup，我们可以调用button.Clicked.Broadcast()多次，并且每次都调用三个处理程序。这是channel不太容易做到的，因此是利用Cond类型的主要原因之一。

与sync包中所包含的大多数其他东西一样，Cond的使用最好被限制在一个紧凑的范围中，或者是通过封装它的类型来暴露在更大范围内。

once

你认为这段代码会输出什么？

```
var count int

increment := func() {
    count++
}
var once sync.Once

var increments sync.WaitGroup
increments.Add(100)
for i:=0;i<100;i++{
    go func() {
        defer increments.Done()
        once.Do(increment)
    }()
}

increments.Wait()
fmt.Printf("Count is %d\n", count)
```

很容易认为结果将是Count is 100，但我肯定你已经注意到了sync.Once变量，在某种程度上通过Do方法把调用增加了一次。事实上，这段代码将打印以下内容：

```
Count is 1
```


顾名思义，sync.Once 是一种类型，它在内部使用一些 sync 原语，以确保即使在不同的 goroutine 上，也只会调用一次 Do 方法处理传递进来的函数。这确实是因为我们将调用 sync.Once 方式执行 Do 方法。

把这种函数只能调用一次的功能放入标准包中似乎是件很奇怪的事情，但事实证明，这种需求经常出现。为了好玩，让我们检查 Go 语言的标准库，看看它自己使用这个原语的频率。下面是一个 grep 命令，它将执行搜索：

```
grep -ir sync.Once $(go env GOROOT)/src |wc -l
```

输出如下：

```
70
```

使用 sync.Once 有几件事需要注意。让我们看另一个例子，你认为它会打印什么？

```
var count int
increment := func() {count++ }
decrement := func() { count-- }

var once sync.Once
once.Do(increment)
once.Do(decrement)

fmt.Printf("Count: %d\n", count)
```

输出如下：

```
Count: 1
```

令人惊讶的是，输出显示的是 1，而不是 0？这是因为 sync.Once 只计算调用 Do 方法的次数，而不是多少次唯一调用 Do 方法。这样，sync.Once 的副本与所要调用的函数紧密耦合，我们再次看到如何在一个严格的范围内合理使用 sync 包中的类型以发挥最佳效果。我建议你通过将 sync.Once 包装在一个小的语法块中来形式化这种耦合：要么是一个小函数，要么是将两者包装在一个结构体中。这个例子你认为会发生什么？

```
var onceA, onceB sync.Once
var initB func()
```

```
initA := func() { onceB.Do(initB) }
initB = func() { onceA.Do(initA) }❶
onceA.Do(initA) ❷
```

❶ 这个调用在 ❷ 返回之前不能进行。

这个程序将会死锁，因为在 ❶ 调用的 Do 直到 ❷ 调用 Do 并退出后才会继续，这是死锁的典型例子。对一些人来说，这可能有点违反直觉，因为它看起来好像我们使用的 sync.Once 是为了防止多重初始化，但 sync.Once 唯一能保证的是你的函数只被调用一次。有时，这是通过死锁程序和暴露逻辑中的缺陷来完成的，在这个例子中是一个循环引用。

池

池（Pool）是 Pool 模式的并发安全实现。关于 Pool 模式的完整解释最好参考设计模式[注2]的文献。不过，由于 Pool 在 sync 软件包中，我们将简要讨论为什么你可能有兴趣使用它。

在较高的层次上，Pool 模式是一种创建和提供可供使用的固定数量实例或 Pool 实例的方法。它通常用于约束创建昂贵的场景（如数据库连接），以便只创建固定数量的实例，但不确定数量的操作仍然可以请求访问这些场景。对于 Go 语言的 sync.Pool，这种数据类型可以被多个 goroutine 安全地使用。

Pool 的主接口是它的 Get 方法。当调用时，Get 将首先检查池中是否有可用的实例返回给调用者，如果没有，调用它的 new 方法来创建一个新实例。当完成时，调用者调用 Put 方法把工作的实例归还到池中，以供其他进程使用。这里有一个简单的例子来说明：

```
myPool := &sync.Pool{
    New: func() interface{} {
        fmt.Println("Creating new instance.")
        return struct{}{}
    },
}
```

注 2：就我个人而言，我推荐 O'Reilly 的优秀书籍《Head First 设计模式》。

```
myPool.Get()❶
instance := myPool.Get()❶
myPool.Put(instance) ❷
myPool.Get() ❸
```

❶ 在这里调用 Pool 的 get 方法。这些调用将执行 Pool 中定义的 new 函数，因为实例还没有实例化。

❷ 我们将先前检索到的实例放在池中，这就增加了实例的可用数量。

❸ 在执行此调用时，我们将重用以前分配的实例并将其放回池中。New 将不会被调用。

我们只看到两个对 New 函数的调用：

```
Creating new instance.
Creating new instance.
```

那么，为什么要使用 Pool，而不只是在运行时实例化对象呢？ Go 语言是有 GC 的，因此实例化的对象将被自动清理。有什么意义？考虑下面这个例子：

```
var numCalcsCreated int
calcPool := &sync.Pool {
    New: func() interface{} {
        numCalcsCreated += 1
        mem := make([]byte, 1024)
        return &mem    ❶
    },
}

// 用 4KB 初始化 pool
calcPool.Put(calcPool.New())
calcPool.Put(calcPool.New())
calcPool.Put(calcPool.New())
calcPool.Put(calcPool.New())

const numWorkers = 1024*1024
var wg sync.WaitGroup
wg.Add(numWorkers)
for i := numWorkers; i > 0; i-- {
    go func() {
       defer wg.Done()
```

```
            mem := calcPool.Get().(*[]byte) ❷
            defer calcPool.Put(mem)

            // 做一些有趣的假设，但是很快就会用这个内存完成
        }()
    }

    wg.Wait()
    fmt.Printf("%d calculators were created.", numCalcsCreated)
```

❶ 注意，我们正在存储 bytes 切片的地址。

❷ 我们断言类型是一个指向 bytes 切片的指针。

输出如下：

```
8 calculators were created.
```

如果我没有用 `sync.Pool` 运行这个例子，尽管结果是不确定的，在最坏的情况下，我可能尝试分配一个十亿字节的内存，但是正如你从输出看到的，我只分配了 4 KB。

另一种常见的情况是，用 Pool 来尽可能快地将预先分配的对象缓存加载启动。在这种情况下，我们不是试图通过限制创建的对象的数量来节省主机的内存，而是通过提前加载获取引用到另一个对象所需的时间，来节省消费者的时间。这在编写高吞吐量网络服务器时十分常见，服务器试图快速响应请求。让我们来看看这样的场景。

首先，让我们创建一个模拟创建到服务的连接的函数。我们会让这次连接花很长时间：

```
func connectToService() interface{} {
    time.Sleep(1*time.Second)
    return struct{}{}
}
```

接下来，让我们了解一下，如果服务为每个请求都启动一个新的连接，那么网络服务的性能如何。我们将编写一个网络处理程序，为每个请求都打开一个新的连接。为了使基准测试简单，我们只允许一次连接：

```
func startNetworkDaemon() *sync.WaitGroup {
    var wg sync.WaitGroup
    wg.Add(1)
    go func() {
        server, err := net.Listen("tcp", "localhost:8080")
        if err!=nil {
             log.Fatalf("cannot listen: %v", err)
        }
        defer server.Close()

        wg.Done()

        for {
           conn, err := server.Accept()
           if err!=nil{
               log.Printf("cannot accept connection: %v", err)
               continue
           }
           connectToService()
           fmt.Fprintln(conn, "")
           conn.Close()
        }
    }()
    return &wg
}
```

现在我们的基准如下：

```
func init() {
    daemonStarted := startNetworkDaemon()
    daemonStarted.Wait()
}

func BenchmarkNetworkRequest(b *testing.B) {
    for i:=0;i<b.N;i++ {
        conn, err := net.Dial("tcp", "localhost:8080")
        if err!=nil {
             b.Fatalf("cannot dial host: %v", err)
        }
        if _, err := ioutil.ReadAll(conn); err != nil {
             b.Fatalf("cannot read: %v", err)
        }
        conn.Close()
    }
}

cd src/gos-concurrency-building-blocks/the-sync-package/pool/ && \
go test -benchtime=10s -bench=.
```

输出如下：

```
BenchmarkNetworkRequest-8      10                          1000385643      ns/op
PASS
Ok                             command-line-arguments      11.008s
```

大概是 1E9 ns/op。这在性能上似乎是合理的，但是我们看看是否可以通过使用 sync.Pool 来改进我们的虚拟服务：

```
func warmServiceConnCache() *sync.Pool {
    p := &sync.Pool {
        New: connectToService,
    }
    for i:=0;i<10;i++ {
        p.Put(p.New())
    }
    return p
}

func startNetworkDaemon() *sync.WaitGroup {
    var wg sync.WaitGroup
    wg.Add(1)
    go func() {
        connPool := warmServiceConnCache()
        server, err := net.Listen("tcp", "localhost:8080")
        if err!=nil {
             log.Fatalf("cannot listen: %v", err)
        }
        defer server.Close()

        wg.Done()

        for {
             conn, err := server.Accept()
             if err!=nil {
                      log.Printf("cannot accept connection: %v", err)
                      continue
             }
             svcConn := connPool.Get()
             fmt.Fprintln(conn, "")
             connPool.Put(svcConn)
             conn.Close()
        }
    }()
    return &wg
}
```

如果我们对它进行基准测试，比如：

```
cd src/gos-concurrency-building-blocks/the-sync-package/pool && \
go test -benchtime=10s -bench=.
```

我们获得如下结果：

```
BenchmarkNetworkRequest-8      5000                    2904307  ns/op
PASS
ok                             command-line-arguments  32.647s
```

2.9E6 ns/op，可以快三个数量级！你可以看到在处理代价昂贵的事务时使用这种模式可以极大地提高响应时间。

正如我们所看到的，你的并发进程需要请求一个对象，但是在实例化之后很快地处理它们时，或者在这些对象的构造可能会对内存产生负面影响，这时最好使用 Pool 设计模式。

然而，有些情况下要谨慎决定你是否应该使用 Pool：如果你使用 Pool 代码所需要的东西不是大概同质的，那么从 Pool 中转化检索到所需要的内容的时间可能比重新实例化内容要花费的时间更多。例如，如果你的程序需要随机和可变长度的切片，那么 Pool 将不会对你有多大帮助。你直接从 Pool 中获得一个正确的切片的概率是很低的。

所以当你使用 Pool 工作时，记住以下几点：

- 当实例化 `sync.Pool`，使用 `new` 方法创建一个成员变量，在调用时是线程安全的。
- 当你收到一个来自 `Get` 的实例时，不要对所接收的对象的状态做出任何假设。
- 当你用完了一个从 Pool 中取出来的对象时，一定要调用 `Put`，否则，Pool 就无法复用这个实例了。通常情况下，这是用 `defer` 完成的。
- Pool 内的分布必须大致均匀。

channel

channel 是由 Hoare 的 CSP 派生的同步原语之一。虽然它们可以用来同步内存访问，但它们最好用于在 goroutine 之间传递信息。正如我们在第 2 章“Go

语言的并发哲学”中所讨论的，在任何大小的程序中，channel 都非常有用，因为它们可以组合在一起。在我介绍 channel 这一节之后，我们将在本章后面“select 语句”中探索该组合。

就像河流一样，一个 channel 充当着信息传送的管道，值可以沿着 channel 传递，然后在下游读出。出于这个原因，我通常用“Stream”来做 chan 变量名的后缀。当你使用 channel 时，你会将一个值传递给一个 chan 变量，然后你程序中的某个地方将它从 channel 中读出。程序中不同的部分不需要相互了解，只需要在 channel 所在的内存中引用相同的位置即可。这可以通过对程序上下游的 channel 引用来完成。

创建一个 channel 非常简单。这里有一个例子，它将一个 channel 的创建扩展到它的声明和后续的实例化中，这样你就可以看到它们到底是怎样的。与 Go 语言中的其他值一样，你可以使用 := 操作符在一个语句中创建 channel，但因为你需要经常声明 channel，因此将这两步操作拆分为单个语句是很有用的：

```
var dataStream chan interface{}❶
dataStream = make(chan interface{})❷
```

❶ 声明一个 channel。因为我们声明的类型是空接口，所以说它的类型是 interface{}。

❷ 使用内置的 make 函数实例化 channel。

这个例子定义了一个 channel（dataStream），任何值都可以写入或读取（因为我们使用了空接口）。channel 也可以声明为只支持单向的数据流，也就是说，可以定义一个 channel 只支持发送或接收信息。我将在后面解释为什么这一点很重要。

要声明一个单向 channel，只需包含 <- 操作符。要声明和实例化一个只能读取的 channel，将 <- 操作符放在左侧，就像这样：

```
var dataStream <-chan interface{}
dataStream := make(<-chan interface{})
```

要声明并创建一个只能发送的 channel，将 <- 操作符放在右侧，就像这样：

```
var dataStream chan<- interface{}
dataStream := make(chan<- interface{})
```

你通常不会看到单向 channel 实例化，但是会经常看到它们用作函数参数和返回类型，和我们看到的一样，这是非常有用的。因为当需要时，Go 语言会隐式地将双向 channel 转换为单向 channel。这里有一个例子：

```
var receiveChan <-chan interface{}
var sendChan chan<- interface{}
dataStream := make(chan interface{})

// 有效的语法：
receiveChan = dataStream
sendChan = dataStream
```

请记住 channel 已经被赋予了类型。在这个例子中，我们创建了一个 `interface{}` 类型 chan 变量，这意味着我们可以将任何类型的数据传递给它，但是我们也可以给它一个更严格的类型来约束它可以传递的数据类型。这是一个整数 channel 的例子，我会用之前已经介绍过的更规范的方式来实例化 channel：

```
intStream := make(chan int)
```

为了使用 channel，我们将再次使用 <- 操作符。通过将 <- 操作符放到 channel 的右边实现发送操作，通过将 <- 操作符放到 channel 的左边实现接收操作。另一种思考方式是数据流向箭头所指方向的变量。让我们看一个简单的例子：

```
stringStream := make(chan string)
go func() {
     stringStream <- "Hello channels!" ❶
}()
fmt.Println(<-stringStream) ❷
```

❶ 我们将字符串文本传递到 `stringStream` channel。

❷ 我们读取 channel 的字符串字面量并将其打印到 `stdout`。

输出如下：

```
Hello channels!
```

很简单，对吧？你所需要的只是一个 channel 变量，你可以将数据传递给它并读取它的数据，但是，尝试将一个值写入只读的 channel 是错误的，并且从只可以写的 channel 读取值也是错误的。如果我们尝试编译下面的例子，Go 语言的编译器会告诉我们操作是非法的：

```
writeStream := make(chan<- interface{})
readStream := make(<-chan interface{})

<-writeStream
readStream <- struct{}{}
```

这里会输出一个异常：

```
invalid operation: <-writeStream (receive from send-only type
   chan<- interface {})
invalid operation: readStream <- struct {} literal (send to receive-only
   type <-chan interface {})
```

这是 Go 语言的类型系统的一部分，它允许我们在处理并发原语时使用 type-safety。正如我们稍后将在本节中看到的，这是一种强大的方法，可以声明我们的 API 并构建可组合的、易于推理的逻辑程序。

回想一下，在本章前面的部分中我们强调了一个事实，那就是 goroutine 是被动调度的，没有办法保证它会在程序退出之前运行。但是前面的示例是完整的，没有省略任何代码。你可能想知道为什么匿名的 goroutine 在 main goroutine 之前就完成运行了，是我运气好吗？让我们简短地讨论一下这个问题。

这个例子起作用了，因为 Go 语言中的 channel 是阻塞的。这意味着只有 channel 内的数据被消费后，新的数据才能写入，而任何试图从空 channel 读取数据的 goroutine 将等待至少一条数据被写入 channel 后才能读到。在这个例子中，`fmt.Println` 会从 `stringStream` 这个 channel 中消费一条数据，它会等 channel 中有数据后才开始消费。同样，匿名 goroutine 试图往 `stringStream` 里写入一条字符串，所以在写入成功之前 goroutine 将不会退出。因此，main goroutine 和匿名 goroutine 一定是阻塞住的。

如果不正确地构造程序，这会导致死锁。请看下面的示例，它引入了一个无意义的条件，以防止匿名 goroutine 往 channel 里写入值：

```
stringStream := make(chan string)
go func() {
       if 0!=1 {  ❶
          return
       }
       stringStream <- "Hello channels!"
}()
fmt.Println(<-stringStream)
```

❶ stringStream channel 并没有值被写入成功。

这里会抛出 panic：

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan receive]:
main.main()
    /tmp/babel-23079IVB/go-src-230795Jc.go:15 +0x97
exit status 2
```

main goroutine 正在等待一个值被写入 stringStream channel，但是由于我们的逻辑，这将永远不会发生。当匿名的 goroutine 退出时，检测到所有 goroutines 都没有运行，并报了一个死锁。在本章的后面将讲解如何构造程序才能做到简单的防止这种死锁，在下一章中会讲解如何完全避免这些问题。在此期间，让我们回到 channel 消费问题上。

通过 <- 操作符的接受形式也可以选择返回两个值，例如：

```
stringStream := make(chan string)
go func() {
       stringStream <- "Hello channels!"
}()
salutation, ok := <-stringStream  ❶
fmt.Printf("(%v): %v", ok, salutation)
```

❶ 在这里会返回一个 string 类型的 salutation 值，一个布尔类型的值，ok。

输出如下：

```
(true): Hello channels!
```

很好奇！布尔值表示什么？第二个返回值是读取操作的一种方式，用于表示该 channel 上有新数据写入，或者是由 closed channel 生成的默认值。稍等，closed channel，那是什么？

在程序中，能够提示 channel 中是否会有新的值写入是非常有用的。这有助于下游的程序知道什么时候消费、退出、给新的 channel 重新建立连接等。我们可以给每个这样的类型一个特殊标记，但这将覆盖大多数开发人员的代码，channel 只是一个简单的数据传输 channel，而不是数据类型的函数，所以关闭 channel 是一个比较普通的操作，就好比哨兵说，“嘿，上游不会写入任何有价值的数据了，想干什么就干什么吧。”我们使用 close 关键字关闭一个 channel，例如：

```
valueStream := make(chan interface{})
close(valueStream)
```

有趣的是，我们也可以从一个已关闭的 channel 读取数据。看这个例子：

```
intStream := make(chan int)
close(intStream)
integer, ok := <- intStream ❶
fmt.Printf("(%v): %v", ok, integer)
```

❶ 从已经关闭的数据流中读取了数据。

输出如下：

```
(false): 0
```

注意，我们从来没有把任何数据推送到 channel 上，立即关闭它。我们仍然能够执行读取操作，事实上，尽管 channel 已经关闭，我们仍然可以继续在这个 channel 上执行读取操作。这是为了支持一个 channel 有单个上游写入，有多个下游读取（在第 4 章中，我们将看到这是一个常见的场景）。第二个返回值（也

就是变量 ok 的值）是 false，这表示我们收到的值是 int 或是 0，而不是推到 stream 上的值。

这为我们提供了一些新的模式。第一个是从 channel 中获取。通过 range 关键作为参数遍历（与 for 语句一起使用），并且在 channel 关闭时自动中断循环。这允许对 channel 上的值进行简洁的迭代。让我们看一个例子：

```
intStream := make(chan int)
go func() {
    defer close(intStream)  ❶
    for i:=1;i<=5;i++ {
        intStream <- i
    }
}()

for integer := range intStream { ❷
    fmt.Printf("%v ", integer)
}
```

❶ 我们确保在 goroutine 退出之前 channel 是关闭的。这是一个很常见的模式。

❷ 遍历了 intStream。

如你所见，所有的值都打印出来了，然后程序退出：

```
1 2 3 4 5
```

注意该循环不需要退出条件，并且 range 方法不返回第二个布尔值。处理一个已关闭的 channel 的细节可以让你保持循环简洁。

关闭 channel 也是一种同时给多个 goroutine 发信号的方法。如果有 n 个 goroutine 在一个 channel 上等待，而不是在 channel 上写 n 次来打开每个 goroutine，你可以简单地关闭 channel。由于一个被关闭的 channel 可以被无数次读取，所以不管有多少 goroutine 在等待它，关闭 channel 都比执行 n 次更适合，也更快。这里有一个例子，可以同时打开多个 goroutine：

```
begin := make(chan interface{})
var wg sync.WaitGroup
for i:=0;i<5;i++ {
    wg.Add(1)
    go func(i int) {
```

```
            defer wg.Done()
            <-begin ❶
            fmt.Printf("%v has begun\n", i)
        }(i)
    }
    fmt.Println("Unblocking goroutines...")
    close(begin) ❷
    wg.Wait()
```

❶ goroutine 会一直等待，直到它被告知可以继续。

❷ 关闭 channel，从而同时打开所有的 goroutine。

你可以看到，在我们关闭开始 channel 之前，所有的 goroutine 都没有开始运行：

```
Unblocking goroutines...
4 has begun
2 has begun
3 has begun
0 has begun
1 has begun
```

请记住在本章前面“sync 包”中，我们讨论了使用 `sync.Cond` 类型执行相同的行为。你当然可以使用它，但是正如我们已经讨论过的， channel 是可组合的，这是我最喜欢的一种在同一时间打开多个 goroutine 的方法。

我们还可以创建 *buffered channel*，它是在实例化时提供容量的 channel。这意味着即使没有在 channel 上执行读取操作，goroutine 仍然可以执行 n 写入，其中 n 是缓冲 channel 的容量。下面是如何声明和实例化一个：

```
var dataStream chan interface{}
dataStream = make(chan interface{}, 4) ❶
```

❶ 创建一个有 4 个容量的缓冲 channel。这意味着我们可以把 4 个东西放到 channel 上，不管它是否被读取。

再一次，我将实例化分解成两行，这样就可以看到一个缓冲的 channel 的声明与一个没有缓冲的 channel 没有什么不同。这有点意思，因为它意味着 goroutine 可以控制实例化一个 channel 时否需要缓冲。这表明，创建一个

channel 应该与 goroutines 紧密耦合，而 goroutines 将会在它上面执行写操作，这样我们就可以更容易地推断它的行为和性能。我们稍后将讨论这个问题。

没有缓冲的 channel 也被定义为缓冲 channel，一个无缓冲 channel 只是一个以 0 的容量创建的缓冲 channel。下面是两个具有等效功能的 channel 示例：

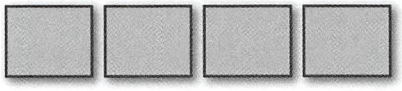
```
a := make(chan int)
b := make(chan int, 0)
```

这两个 channel 都是具有零容量的 int channel。请记住，当我们讨论阻塞时，如果说 channel 是满的，那么写入 channel 阻塞，如果 channel 是空的，则从 channels 读取的是什么？“Full”和“empty”是容量或缓冲区大小的函数。无缓冲 channel 的容量为零，因此在任何写入之前 channel 已经满了。一个没有下游接受的容量为 4 的缓冲 channel 在被写 4 次之后就满了，并且在写第 5 次的时候阻塞，因为它没有其他地方放置第五个元素。与未缓冲的 channel 一样，缓冲 channel 仍然阻塞；channel 为空或满的前提条件是不同的。通过这种方式，缓冲 channel 是一个内存中的 FIFO 队列，用于并发进程进行通信。

为了帮助理解这一点，让我们用例子来解释一个具有 4 个容量的缓冲 channel 的情况。首先，让我们来初始化：

```
c := make(chan rune, 4)
```

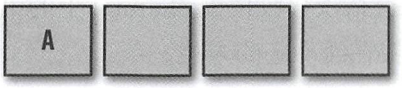
从逻辑上讲，这创建了一个带有四个槽的缓冲区，比如：



现在让我们往 channel 里写数据：

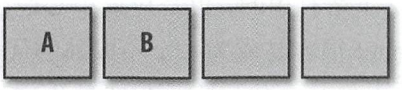
```
c<-'A'
```

当这个 channel 没有下游读取时，一个数据将被放置在 channel 缓冲区的第一个槽中，就像这样：

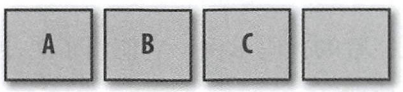


每个后续的值写入到缓冲 channel（假设没有下游读取）会在缓冲 channel 填充剩余的槽，比如：

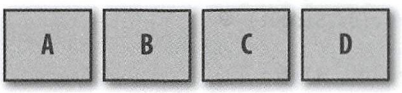
```
c<-'B'
```




```
c<-'C'
```

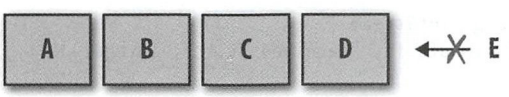



```
c<-'D'
```



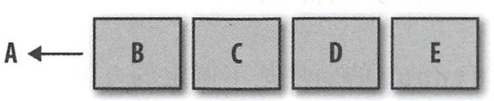
经过 4 次写入，我们的缓冲 channel 容量为 4。如果我们试图再次写入 channel 会发生什么？

```
c<-'E'
```



执行操作的 goroutine 被阻塞了 !goroutine 将会被阻塞，直到有一些 goroutine 执行读取了在缓冲区的数据。让我们看看这是什么情况：

```
<-c
```



如你所见，下游去读取会依次接收位于 channel 上的第一个数据 A，数据写入是阻塞的操作，E 被放置在缓冲区的末尾。

它还提到，如果一个缓冲 channel 是空的，并且有一个下游接收，那么缓冲区将被忽略，并且该值将直接从发送方传递到接收方。在实践中这是透明的，但是对了解缓冲 channel 的配置是值得的。

缓冲 channel 在某些情况下是有用的，但是应该小心地创建它们。在下一章中，我们将看到，缓冲 channel 很容易成为一个不成熟的优化，并且使隐藏的死锁更不容易发生。这听起来像是一件好事，但我猜你宁愿在第一次写代码的时候发现死锁，而不是在生产系统崩溃的时候才发现。

让我们来看看另一个更完整的代码示例，它使用了缓冲 channel，这样就可以更好地了解它们的工作原理：

```
var stdoutBuff bytes.Buffer  ❶
defer stdoutBuff.WriteTo(os.Stdout) ❷

intStream := make(chan int, 4) ❸
go func() {
    defer close(intStream)
    defer fmt.Fprintln(&stdoutBuff, "Producer Done.")
    for i:=0;i<5;i++ {
        fmt.Fprintf(&stdoutBuff, "Sending: %d\n", i)
        intStream <- i
    }
}()

for integer := range intStream {
    fmt.Fprintf(&stdoutBuff, "Received %v.\n", integer)
}
```

❶ 创建一个内存缓冲区，以帮助减少输出的不确定性。它没有给我们任何保证，但它比直接写 stdout 要快一点。

❷ 确保在进程退出之前缓冲区内容需要被写入到 stdout。

❸ 创建了具有一个容量的缓冲 channel。

在本例中，输出到 stdout 的顺序是不确定的，但仍然可以大致了解匿名 goroutine 是如何工作的。如果看一下输出，你就会发现我们的匿名 goroutine

是如何把它的 5 个结果都写到 intStream 上的，然后在 main goroutine 将每个结果都推送出去之前就退出：

```
Sending: 0
Sending: 1
Sending: 2
Sending: 3
Sending: 4
Producer Done.
Received 0.
Received 1.
Received 2.
Received 3.
Received 4.
```

这是一个适合某些条件的优化例子：如果 goroutine 写入一个 channel 的时候会知道它需要写多少条数据，它可以创建一个容量是写数量缓冲 channel 的，然后尽快往 channel 里写数据。当然，我们将会在下一章中讨论这么做的注意事项。

我们讨论了无缓冲 channel、缓冲 channel、双向 channel 和单向 channel。我们没有覆盖的 channel 的唯一方面是 channel 的默认值 nil。程序如何与值为 nil 的 channel 交互？首先，让我们试着从 nil channel 中读取数据：

```
var dataStream chan interface{}
<-dataStream
```

输出如下：

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan receive (nil chan)]:
main.main()
     /tmp/babel-23079IVB/go-src-23079O4q.go:9 +0x3f
exit status 2
```

死锁！这表明从 nil channel 读取数据将阻塞（尽管不一定是死锁）程序。那如果写入会发生什么呢？

```
var dataStream chan interface{}
dataStream <- struct{}{}
```

输出如下：

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send (nil chan)]:
main.main()
     /tmp/babel-23079IVB/go-src-23079dnD.go:9 +0x77
exit status 2
```

它看起来像即使写到 nil channel 也会阻塞。只有一个操作可用，关闭。如果我们试图关闭 nil channel 会发生什么？

```
var dataStream chan interface{}
close(dataStream)
```

输出如下：

```
panic: close of nil channel

goroutine 1 [running]:
panic(0x45b0c0, 0xc42000a160)
   /usr/local/lib/go/src/runtime/panic.go:500 +0x1a1
main.main()
   /tmp/babel-23079IVB/go-src-230794uu.go:9 +0x2a
exit status 2
```

这可能是在 nil channel 上执行的所有操作中最糟糕的结果 panic。确保你所使用的 channel 都会被初始化。

我们已经讨论了很多如何与 channel 交互的规则。现在你已经了解在 channel 上执行操作的方式和原因，让我们为使用 channel 的创建一个可参考方式。表 3-2 列举了 channel 上的操作，以及在使用 channel 状态下会发生什么。

表 3-2：channel 操作的结果给出了 channel 的状态

操作	Channel 状态	结果
Read	nil	阻塞
	打开且非空	输出值
	打开但空	阻塞
	关闭的	< 默认值 >, false
	只写	编译错误

表 3-2：channel 操作的结果给出了 channel 的状态（续）

操作	Channel 状态	结果
Write	nil	阻塞
	打开的但填满	阻塞
	打开的且不满	写入值
	关闭的	**panic**
	只读	编译错误
close	nil	**panic**
	打开且非空	关闭 Channel；读取成功，直到通道耗尽，然后读取产生值的默认值
	打开但空	关闭 Channel；读到生产者的默认值
	关闭的	**panic**
	只读	编译错误

如果检查这张表，我们会看到一些可能导致麻烦的数据。我们有三种操作可以导致 goroutine 阻塞，三种操作会导致程序 panic! 乍一看，似乎 channel 可能会有问题，但在研究了具体执行结果并确定了 channel 的使用方式之后，它就变得不那么可怕了，并且开始变得有意义了。让我们看看如何组织不同类型的 channel 来构建健壮和稳定的东西。

我们应该做的第一件事是在正确的环境中配置 channel，即分配 channel 所有权。我将把所有权定义为实例化、写入和关闭 channel 的 goroutine。就像没有 GC 的语言的内存一样，重要的是要弄清楚哪个 goroutine 拥有 channel，以便从逻辑上推演我们的程序。单向 channel 声明的是一种工具，它将允许我们区分 channel 的拥有者和 channel 的使用者：channel 所有者对 channel（chan 或 chan<-）有一个写访问视图，而 channel 使用者只对 channel 有一个只读视图（<-chan）。一旦我们将 channel 所有者和非 channel 所有者区分开来，前面的表的结果自然就会很清晰，我们可以开始将责任分配给那些拥有 channel 的 goroutine 和不拥有 channel 的 goroutine。

让我们从 channel 的所有者开始。拥有 channel 的 goroutine 应该具备如下：

1. 实例化 channel。
2. 执行写操作，或将所有权传递给另一个 goroutine。

3. 关闭 channel。
4. Ecapsulate 在此列表中的前三件事，并通过一个只读 channel 将它们暴露出来。

通过将这些责任分配给 channel 所有者，一些事情发生了：

- 因为我们初始化了 channel，所以我们将死锁的风险转移到 `nil` channel 上。
- 因为我们初始化了 channel，所以我们通过关闭一个 `nil` channel 来消除 panic 的风险。
- 因为我们决定了 channel 何时关闭，所以我们通过写入一个关闭的 channel 来消除 panic。
- 因为我们决定了 channel 何时关闭，所以我们不止一次关闭 channel，从而消除了 panic 的风险。
- 我们在编译时使用类型检查器，以防止写入 channel 异常。

现在让我们看看在读取 channel 时可能发生的阻塞操作。作为一个 channel 的消费者，我只需要担心两件事：

- 知道 channel 是何时关闭的。
- 正确的处理阻塞。

为了解决第一个问题，我们只需像之前说的那样从 read 操作中检查第二个返回值。第二点更难定义，因为它取决于你的算法，可能想要超时，可能想要停止消费，或者可能只是对阻塞进程的生命周期有需求。重要的是，作为一个消费者，应该知道读取是阻塞的事实。我们将在接下来的章节中研究如何优雅的实现 channel 消费。

现在，让我们看一个例子来帮助阐明这些概念。让我们创建一个拥有 channel 的 goroutine，以及一个处理 channel 阻塞和关闭的消费者：

```
chanOwner := func() <-chan int {
```

```
    resultStream := make(chan int, 5) ❶
    go func() { ❷
        defer close(resultStream) ❸
        for i:=0;i<=5;i++ {
            resultStream <- i
        }
    }()
    return resultStream  ❹
}

resultStream := chanOwner()
for result := range resultStream { ❺
    fmt.Printf("Received: %d\n", result)
}
fmt.Println("Done receiving!")
```

❶ 实例化一个缓冲 channel。因为知道将产生 6 个结果，我们创建一个有 5 个缓冲的 channel，这样 goroutine 就能尽快完成。

❷ 启动一个匿名的 goroutine，它在 `resultStream` 上执行写操作。注意，我们已经颠倒了如何创建 goroutine。它现在被封装在外围函数中。

❸ 我们确保一旦执行完成 `resultStream` 就会关闭。作为 channel 所有者，这是我们必须做的。

❹ 在这里我们返回 channel。由于返回值被声明为一个只读 channel，因此 `resultStream` 将隐式地转换为只读消费者。

❺ 遍历 `resultStream`。作为消费者，我们只关心阻塞和 channel 的关闭。

输出如下：

```
Received: 0
Received: 1
Received: 2
Received: 3
Received: 4
Received: 5
Done receiving!
```

注意，`resultStream` channel 的生命周期是如何封装在 chan 所有者函数中的。很明显，写入不会在 nil 或已关闭的 channel 上发生，而且关闭总是只会发生一次。这从我们的程序中消除了大量的风险。我强烈建议在你的程序中，尽量保持 channel 所有权的范围很小，这样事情就变得显而易见了。如果有一

个 channel 作为结构体的成员变量，并且有许多方法，它将很快变得不清楚该 channel 的行为方式。

消费者函数只能执行 channel 的读取方法，因此只需要知道它应该如何处理阻塞读取和 channel 的关闭。在这个小示例中，我们已经采取了这样的操作：在 channel 关闭之前，阻塞程序是完全可以运行的。

如果你设计让你的代码遵循这一原则，那么你的系统就会变得更容易梳理，并且它也更有可能按照你的期望来执行。我不能保证你永远不会引入死锁或 panic，但当你这么做的时候，我想你会发现你的 channel 所有权的范围已经变得太大了，或者所有权已经变得不清晰了。

channel 是吸引人们使用 Go 语言的原因之一。结合了 goroutine 和闭包的简单性，我很清楚地知道编写干净、正确的并发代码是多么容易。在很多方面，channel 是将 goroutine 黏合在一起的黏合剂。本章应该给了你一个关于什么是 channel 以及如何使用它们的很好的概述。真正的乐趣始于我们开始编写 channel 以形成高阶并发设计模式。我们会在下一章讲到。

select 语句

`select` 语句是将 channel 绑定在一起的黏合剂，这就是我们如何在一个程序中组合 channel 以形成更大的抽象事务的方式。如果 channel 是将 goroutine 连接在一起的黏合剂，那么声明 select 的语句是做什么的呢？声明 select 语句是一个具有并发性的 Go 语言程序中最重要的事情之一，这并不是夸大其词。在一个系统中两个或多个组件的交集中，可以在本地、单个函数或类型以及全局范围内找到 select 语句绑定在一起的 channel。除了连接组件之外，在程序中的这些关键节点上，select 语句可以帮助安全地将 channel 与诸如取消、超时、等待和默认值之类的概念结合在一起。

相反，如果 `select` 语句是程序的通用语言，它们只处理 channel，那么程序的组件应该如何协调？我们将在第 5 章中专门研究这个问题（提示，更推荐使用 channel）。

那么这些强大的 select 语句是什么呢？我们如何使用它们，它们是如何工作的？让我们先把它放出来。这里有一个很简单的例子：

```
var c1, c2 <-chan interface{}
var c3 chan<- interface{}
select {
case <- c1:
    // 执行某些逻辑
case <- c2:
    // 执行某些逻辑
case c3<- struct{}{}:
    // 执行某些逻辑
}
```

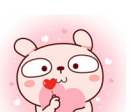
它看起来有点像一个选择模块，不是吗？就像一个选择模块，一个 select 模块包含一系列的 case 语句，这些语句可以保护一系列语句。然而，这就是相似之处。与 switch 块不同，select 块中的 case 语句没有测试顺序，如果没有满足任何条件，执行也不会失败。

相反，所有的 channel 读取和写入都需要查看是否有任何一个已准备就绪可以用的数据[注3]：在读取的情况下关闭 channel，以及写入不具备下游消费能力的 channel。如果所有 channel 都没有准备好，则执行整个 select 语句模块。当一个 channel 准备好了，这个操作就会继续，它相应的语句就会执行。让我们来看一个简单的例子：

```
start := time.Now()
c := make(chan interface{})
go func() {
    time.Sleep(5*time.Second)
    close(c)  ❶
}()

fmt.Println("Blocking on read...")
select {
case <-c:  ❷
    fmt.Printf("Unblocked %v later.\n", time.Since(start))
}
```

❶ 在等待 5s 后关闭 channel。

注 3：实际运行的状况要更复杂一些，我们会在第 6 章提及。



❷ 尝试在 channel 上读取数据。注意，在编写这段代码时，我们不需要 select 语句。可以简单地使用 <-c，但是我们将在这个示例中进行扩展。

输出如下：

```
Blocking on read...
Unblocked 5.000170047s later.
```

如你所见，在进入 select 模块后大约 5 秒，我们就会解锁。这是一种简单而有效的方法来阻止我们等待某事的发生，但如果我们思考一下，我们可以提出一些问题：

- 当多个 channel 有数据可供给下游读取的时候会发生什么？
- 如果没有任何可用的 channel 怎么办？
- 如果我们想要做一些事情，但是没有可用的 channels 怎么办？

多个 channel 同时是可用的这个问题似乎很有趣。让我们试试，看看会发生什么！

```
c1 := make(chan interface{});close(c1)
c2 := make(chan interface{});close(c2)

var c1Count, c2Count int
for i:=1000;i>=0;i-- {
    select {
    case <-c1:
        c1Count++
        case <-c2:
        c2Count++
    }
}

fmt.Printf("c1Count: %d\nc2Count: %d\n", c1Count, c2Count)
```

输出如下：

```
c1Count: 505
c2Count: 496
```


如你所见，在一千次迭代中，大约有一半的时间从 c1 读取 select 语句，大约一半的时间从 c2 读取。这看起来很有趣，也许有点太巧了。事实如此 !Go 语言运行时将在一组 case 语句中执行伪随机选择。这就意味着，在你的 case 语句集合中，每一个都有一个被执行的机会。

乍一看，这似乎并不重要，但背后的原因却非常有趣。让我们先做一个很明显的阐述：Go 语言运行时无法解析 select 语句的意图，也就是说，它不能推断出问题空间，或者说为什么将一组 channel 组合到一个 select 语句中。正因为如此，运行时所能做的最好的事情就是在平均情况下运行良好。一种很好的方法是将一个随机变量引入到等式中（在这种情况下，select 后续的 channel）。通过加权平均每个 channel 被使用的机会，所有使用 select 语句的程序将在平均情况下表现良好。

关于第二个问题：如果没有任何 channel 可用，会发生什么？如果所有的 channel 都被阻塞了，如果没有可用的，但是你可能不希望永远阻塞，可能需要超时机制。Go 语言的 time 包提供了一种优雅的方式，可以在 select 语句中很好地使用 channel。这里有一个例子：

```
var c <-chan int
select {
case <-c:  ❶
case <-time.After(1 * time.Second):
    fmt.Println("Timed out.")
}
```

❶ 这个 case 语句永远不会被解锁，因为我们是从 nil channel 读取的。

输出如下：

```
Timed out.
```

time.After 函数通过传入 time.Duration 参数返回一个数值并写入 channel，该 channel 会返回执行后的时间。这为 select 语句提供了一种简明

的方法。我们将在第 4 章重新讨论这个模式，在这里我们将讨论一个更健壮的解决方案。

最后一个问题：当没有可用 channel 时，我们需要做些什么？像 case 语句一样，select 语句也允许默认的语句。就像“case”语句一样，当“select”语句中的所有 channel 都被阻塞的时候，“select”语句也允许你调用默认语句。以下是一个实例：

```
start := time.Now()
var c1, c2 <-chan int
select {
case <-c1:
case <-c2:
default:
    fmt.Printf( "In default after %v\n\n", time.Since(start))
}
```

输出如下：

```
In default after 1.421µs
```

可以看到，它几乎是瞬间运行了默认语句。这允许在不阻塞的情况下退出 select 模块。通常，你将看到一个默认的子句，它与 for-select 循环一起使用。这允许 goroutine 在等待另一个 goroutine 上报结果的同时，可以继续执行自己的操作。这里有一个例子：

```
done := make(chan interface{})
go func() {
    time.Sleep(5*time.Second)
    close(done)
}()

workCounter := 0
loop:
for {
    select {
    case <-done:
        break loop
    default:
    }
```


```
        // 模拟工作行为
        workCounter++
        time.Sleep(1*time.Second)
    }

    fmt.Printf("Achieved %v cycles of work before signalled to stop.\n", workCounter)
```

输出如下：

```
Achieved 5 cycles of work before signalled to stop.
```

在这种情况下，我们有一个循环，它在执行某种操作，偶尔检查它是否应该被停止。

最后，对于空的 select 语句有一个特殊的情况：选择没有 case 子句的语句。看起来像这样：

```
select {}
```

这个语句将永远阻塞。

在第 6 章中，我们将深入研究 select 语句是如何工作的。从更高层次的角度来看，它应该是显而易见的，它可以帮助你安全高效地组合各种概念和子系统。

GOMAXPROCS 控制

在 runtime 包中，有一个函数称为 GOMAXPROCS。在我看来，这个名称是有误导性的：人们通常认为这个函数与主机上的逻辑处理器的数量有关（而且与它调度方式有关），但实际上这个函数控制的 OS 线程的数量将承载所谓的“工作队列”。有关这个函数的更多信息以及它的工作原理，请参见第 6 章。

在 Go 语言 1.5 之前，GOMAXPROCS 总是被设置为 1，通常你会在大多数 Go 语言程序中找到这段代码：

```
runtime.GOMAXPROCS(runtime.NumCPU())
```


几乎大部分开发人员希望当他们的程序正在运行时，可以充分利用机器上的所有 CPU 核心。因此，在随后的 Go 语言版本中，它自动设置为主机上逻辑 CPU 的数量。

那么为什么要调整这个值呢？大部分时间你都不太想去调节它。Go 语言的调度算法在大多数情况下已经足够好了，在增加或减少工作队列和线程数量的情况下，可能会造成更多的问题，但是仍然有一些情况会改变这个值。

例如，我在一个项目上调试，这个项目有一个测试组件，它被竞争环境困扰。不管怎么说，这个团队有几个包，有时候测试失败。我们运行测试的主机有四个逻辑 CPU，因此在任何一个点上，我们都有四个 goroutines 同时执行。通过增加 GOMAXPROCS 以超过我们拥有的逻辑 CPU 数量，我们能够更频繁地触发竞争条件，从而更快地修复它们。

其他人可能通过实验发现，他们的程序在一定数量的工作队列和线程上运行得更好，但我更主张谨慎些。如果你通过调整这个方法来压缩性能，那么在每次提交之后，当你使用不同的硬件，以及使用不同版本的 Go 语言时，一定要这样做。调整这个值会使你的程序更接近它所运行的硬件，但以抽象和长期性能稳定为代价。

小结

在本章中，我们已经介绍了所有的基本并发原语。如果你已经阅读并理解了这一点，那么恭喜你！你可以很好地在编写具有可读性和逻辑正确的程序。你知道在何时使用 sync 包中的同步原语访问内存，比使用 channel 和 select 语句“通过通信来共享内存”更合适。

在编写并发 Go 语言代码时，需要理解的是如何将这些原语以结构化的方式组合起来，并且易于理解。在这本书的下半部分，我们将讨论如何做到这一点。下一章是关于如何使用社区已经发现的模式组合这些原语。

第 4 章

Go 语言的并发模式

我们已经探索了 Go 语言的并发原语的基本原理，并讨论了如何正确使用这些原语。在本章中，我们将深入探讨如何将这些基元组合成模式，以帮助保持系统的可扩展性和可维护性。

但是，在我们开始之前，我们需要谈谈本章所包含的一些模式的格式。在很多示例中，我们将使用传递空接口（`interface{}`）的 channel。在 Go 语言中使用空接口（`interface{}`）是有争议的，不过，我们出于以下几个原因选择使用了空接口。首先，它使得在书的其余部分编写简洁的例子变得更容易。其次，在某些情况下，我认为这更能代表该模式正在努力达成的目标。我们将在本章后面的“pipeline”直接讨论这一点。

如果这对你来说过于难以接受，请记住你始终可以为此代码创建 Go 语言生成器，并生成模式以利用你感兴趣的类型。

所以，让我们深入了解 Go 语言的一些并发模式吧！

约束

在编写并发代码的时候，有以下几种不同的保证操作安全的方法。我们已经介绍了其中两个：

- 用于共享内存的同步原语（如 `sync.Mutex`）。
- 通过通信共享内存来进行同步（如 channel）。

但是，在并发处理中还有其他几种情况也是隐式并发安全的：

- 不会发生改变的数据。
- 受到保护的数据。

从某种意义上讲，不可变数据是理想的，因为它是隐式地并行安全的。每个并发进程可能对相同的数据进行操作，但不能对其进行修改。如果要创建新数据，则必须创建具有所需修改的数据的新副本。这不仅可以减轻开发人员的认知负担，并且可以使程序运行得更快，这将使程序的临界区减少（或者完全消除临界区）。在 Go 语言中，可以通过编写利用值的副本而不是指向内存值的指针的代码来实现此目的。有些语言支持使用明确不变的值的指针，然而，Go 语言不在其中。“约束”还可以使开发人员减少临界区的长度以及承担更小的认知负担。约束并发值的技术比简单传递值的副本要复杂一点，所以本章我们将深入介绍这些约束技术。

“约束”是一种确保了信息只能从一个并发过程中获取到的简单且强大的方法。达到此目的时，并发程序隐式安全，不需要同步。有两种可能的约束：特定约束和词法约束。特定约束是指通过公约实现约束时，无论是由语言社区、你所在的团队，还是你的代码库设置。在我看来，坚持约束很难在任何规模的项目上进行协调，除非你有工具在每次有人提交代码时对你的代码进行静态分析。下面是一个特定约束的例子，它说明了原因：

```
data := make([]int, 4)

loopData := func(handleData chan<- int) {
    defer close(handleData)
    for i := range data {
        handleData <- data[i]
    }
}

handleData := make(chan int)
go loopData(handleData)

for num := range handleData {
    fmt.Println(num)
}
```


我们可以看到，loopData 函数和 handleData channel 上的循环都可以使用整数的数据切片。然而，按照惯例，我们只能从 loopData 函数访问它。但是，随着代码被更多人所触及，deadline 缩短，就可能会出错，并且约束可能会被打破并导致问题。正如我所提到的，一个静态分析工具可能会遇到这类问题，但对 Go 语言代码库进行静态分析表明一个成熟度水平并不是很多团队达到的。这就是为什么我更喜欢约束这个词，它使用编译器来执行约束。

词法约束涉及使用词法作用域仅公开用于多个并发进程的正确数据和并发原语。这使得做错事是不可能的。实际上我们已经在第 3 章中谈到了这个主题。回想一下 channel 部分，它讨论的只是将 channel 的读或写处理暴露给需要它们的并发进程。我们再来看看这个例子：

```
chanOwner := func() <-chan int {
    results := make(chan int, 5) ❶
    go func() {
        defer close(results)
        for i:=0;i<=5;i++ {
             results <- i
        }
    }()
    return results
}

consumer := func(results <-chan int) {❸
    for result := range results {
        fmt.Printf("Received: %d\n", result)
    }
    fmt.Println("Done receiving!")
}

results := chanOwner()❷
consumer(results)
```

❶ 在 chanOwner 函数的词法范围内实例化 channel。这将结果写入 channel 的处理的范围约束在它下面定义的闭包中。换句话说，它包含了这个 channel 的写入处理，以防止其他 goroutine 写入它。

❷ 收到了 channel 的读处理，能够将它传递给消费者，消费者只能从中读取信息。这又一次将 main goroutine 约束在 channel 的只读视图中。

❸ 收到一个 int channel 的只读副本。通过声明我们要求的唯一用法是读取访问，我们将 channel 内的使用约束为只读。

这样设置，在这个小例子中就不可能利用这些 channel。这是一个很好的读取的方案，但可能不是一个非常有趣的例子，因为 channel 是并发安全的。让我们来看一个使用不是并发安全的数据结构的约束的例子，它是一个 bytes.Buffer 的实例：

```
printData := func(wg *sync.WaitGroup, data []byte) {
    defer wg.Done()

    var buff bytes.Buffer
    for _,b:=range data {
            fmt.Fprintf(&buff, "%c", b)
      }
      fmt.Println(buff.String())
}

var wg sync.WaitGroup
wg.Add(2)
data := []byte("golang")
go printData(&wg, data[:3]) ❶
go printData(&wg, data[3:]) ❷

wg.Wait()
```

❶ 这里我们传入包含数据结构中前三个字节的切片。

❷ 这里我们传入一个包含数据结构中最后三个字节的切片。

在这个例子中，你可以看到，因为 printData 没有在数据切片周围关闭，所以它不能访问它，并且需要占用一部分字节才能操作。我们传递切片的不同子集，因此约束了我们开始的 goroutine，只是我们传入切片的一部分。由于词法范围的原因，我们已经不可能执行错误的操作，所以我们不需要通过通信完成内存访问同步或共享数据[注 1]。

那么有什么意义呢？如果我们有同步功能，为什么要约束？答案是提高了性能并降低了开发人员的认知负担。同步带来了成本，如果你可以避免它，你将不会有任何临界区，因此你不必为同步它们付出任何成本。你也可以通过同步回避所有可能的问题，开发人员根本不必担心这些问题。利用词法约束的并发代码通常比不具有词法约束变量的并发代码更易于理解。这是因为在你的词法范围内，你可以编写同步代码。

注 1：我忽略了通过不安全的软件包手动操作内存的可能性。这是不安全的原因！

话虽如此，建立约束可能很困难，所以有时我们必须回到我们美妙的 Go 语言并发原语。

for-select 循环

在 Go 语言程序中你会一遍又一遍地看到 for-select 循环。它不过是这样的：

```
for { // 要不就无限循环，要不就使用 range 语句循环
      select {
      // 使用 channel 进行作业
      }
}
```

有以下几种情况你可以见到这种模式。

向 channel 发送迭代变量

通常情况下，你需要将可迭代的内容转换为 channel 上的值。这不是什么幻想，通常看起来像这样：

```
for _, s := range []string{"a", "b", "c"} {
    select {
    case <-done:
        return
    case stringStream <- s:
    }
}
```

循环等待停止

创建循环，无限循环直到停止的 goroutine 很常见。这个有一些变化。你选择哪一个纯粹是一种个人爱好。

第一种变体保持 select 语句尽可能短：

```
for {
    select {
    case <-done:
        return
    default:
    }

    // 进行非抢占式任务

}
```

如果已经完成的 channel 未关闭，我们将退出 select 语句并继续执行 for 循环的其余部分。

第二种变体将工作嵌入到选择语句的默认子句中：

```
for {
    select {
    case <-done:
        return
    default:
        // 进行非抢占式任务
    }
}
```

当我们输入 select 语句时，如果完成的 channel 尚未关闭，我们将执行 default 子句。

这种模式没有什么别的了，但它在任何地方都会被用到，所以值得一提。

防止 goroutine 泄漏

正如我们在第 3 章“goroutine”介绍的那样，我们知道 goroutine 廉价且易于创建，这是让 Go 语言这么富有成效的原因之一。运行时将多个 goroutine 复用到任意数量的操作系统线程，以便我们不必担心该抽象级别。但是 goroutine 还是需要消耗资源，而且 goroutine 不会被运行时垃圾回收，所以无论 goroutine 所占用的内存有多么的少，我们都不希望我们的进程对此没有感知。那么我们如何去确保他们被清理干净？

让我们从头开始，逐步思考这个问题：为什么一个 goroutine 需要存在呢？在第 2 章中，我们确定 goroutine 代表可能或不可能相互平行运转的工作单位。goroutine 有以下几种方式被终止：

- 当它完成了它的工作。
- 因为不可恢复的错误，它不能继续工作。
- 当它被告知需要终止工作。

我们可以很简单地使用前两种方法，因为这两种方法就隐含在你的算法中，但是“取消工作”又是怎样工作的呢？由于网络的影响，事实证明这是最重要的一点：如果你开始了一个 goroutine，最有可能以某种有组织的方式与其他几个 goroutine 合作。我们甚至可以将这种相互连接表现为一个图表：子 goroutine 是否应该继续执行可能是以许多其他 goroutine 状态的认知为基础的。

goroutine（通常是 main goroutine）具有这种完整的语境知识应该能够告诉其子 goroutine 终止。我们将在下一章继续研究大规模的 goroutine 的相互依赖关系，但现在让我们考虑如何确保一个子 goroutine 被清理。让我们从一个简单的 goroutine 泄漏开始：

```
doWork := func(strings <-chan string) <-chan interface{} {
    completed := make(chan interface{})
    go func() {
        defer fmt.Println("doWork exited." )
        defer close(completed)
        for s := range strings {
             // 做些有趣的操作
             fmt.Println(s)
            }
    }()
    return completed
}

doWork(nil)
// 也许这里有其他的操作需要进行
fmt.Println( “Done.")
```

在这里，我们看到 main goroutine 将一个空的 channel 传递给了 `doWork`。因此，字符串 channel 永远不会写入任何字符串，并且包含 `doWork` 的 goroutine 将在此过程的整个生命周期中保留在内存中（如果我们在 `doWork` 和 main goroutine 中加入了 goroutine，甚至会死锁）。

在这个例子中，这个过程的生命周期很短，但是在一个真正的程序中，goroutine 可以很容易地在一个长期生命的程序开始时启动。在最糟糕的情况

下，main goroutine 可能会在其生命周期内持续的将其他的 goroutine 设置为自旋，这会导致内存利用率的下降。

成功减轻这种情况的方法是在父 goroutine 和其子 goroutine 之间建立一个信号，让父 goroutine 向其子 goroutine 发出信号通知。按照惯例，这个信号通常是一个名为 done 的只读 channel。父 goroutine 将该 channel 传递给子 goroutine，然后在想要取消子 goroutine 时关闭该 channel。例如：

```
doWork := func(
  done <-chan interface{},
  strings <-chan string,
) <-chan interface{} { ❶
    terminated := make(chan interface{})
    go func() {
        defer fmt.Println("doWork exited.")
        defer close(terminated)
        for {
            select {
            case s := <-strings:
          // 做一些有意思的操作
                fmt.Println(s)
            case <-done:  ❷
                return
            }
        }
    }()
    return terminated
}

done := make(chan interface{})
terminated := doWork(done, nil)

go func() { ❸
    // 在 1 秒钟之后取消本操作
    time.Sleep(1 * time.Second)
    fmt.Println("Canceling doWork goroutine...")
    close(done)
}()

<-terminated ❹
fmt.Println("Done.")
```

❶ 在这里，我们将完成的 channel 传递给 doWork 函数。作为惯例，这个 channel 是第一个参数。

❷ 在这一行上，我们看到了在实际编程中无处不在的 select 模式。我们的一

个案例陈述是检查我们的 done channel 是否已经发出信号。如果有的话，我们从 goroutine 返回。

❸ 在这里我们创建另一个 goroutine，如果超过 1s 就会取消 doWork 中产生的 goroutine。

❹ 这就是我们加入从 main goroutine 的 doWork 中产生的 goroutine 的地方。

然后，输出结果如下：

```
Canceling doWork goroutine...
doWork exited.
Done.
```

你可以看到，尽管我们给我们的字符串 channel 中传递了 `nil`，我们的 goroutine 仍然成功退出。与之前的例子不同，在这个例子中，我们加入了两个 goroutine，但没有造成死锁。这是因为在我们加入两个 goroutine 之前，我们创建了第三个 goroutine 来在 `doWork` 执行 1s 之后取消 `doWork` 中的 goroutine。我们已经成功消除了我们的 goroutine 泄漏！

前面的例子很好地处理了在 channel 上接收 goroutine 的情况，但是如果我们正在处理相反的情况：一个 goroutine 阻塞了向 channel 进行写入的请求？以下是演示此问题的简单示例：

```
newRandStream := func() <-chan int {
    randStream := make(chan int)
    go func() {
        defer fmt.Println("newRandStream closure exited." ) ❶
        defer close(randStream)
        for {
             randStream <- rand.Int()
            }
    }()
    return randStream
}

randStream := newRandStream()
fmt.Println("3 random ints:" )
for i:=1;i<=3;i++ {
    fmt.Printf( “%d: %d\n", i, <-randStream)
}
```

❶ 这里我们在 goroutine 成功终止时打印出一条消息。

运行此代码会产生：

```
3 random ints:
1: 5577006791947779410
2: 8674665223082153551
3: 6129484611666145821
```

你可以从输出中看到 defer 语句中的 `fmt.Println` 语句永远不会运行。在循环的第三次迭代之后，我们的 goroutine 试图将下一个随机整数发送到不再被读取的 channel。我们无法告诉生产者它可以停止。解决方案就像接收案例一样，为生产者 goroutine 提供一个通知它退出的 channel：

```
newRandStream := func(done <-chan interface{}) <-chan int {
    randStream := make(chan int)
    go func() {
        defer fmt.Println("newRandStream closure exited." )
        defer close(randStream)
        for {
             select {
             case randStream <- rand.Int():
             case <-done:
                     return
             }
        }
    }()
    return randStream
}

done := make(chan interface{})
randStream := newRandStream(done)
fmt.Println("3 random ints:" )
for i:=1;i<=3;i++ {
    fmt.Printf("%d: %d\n", i, <-randStream)
}
close(done)

// 模拟正在进行的工作
time.Sleep(1 * time.Second)
```

代码输出：

```
3 random ints:
1: 5577006791947779410
2: 8674665223082153551
3: 6129484611666145821
newRandStream closure exited.
```

我们发现现在 goroutine 已经被正确地清理了。

现在我们知道如何确保 goroutine 不泄漏，我们可以规定一个约定：如果 goroutine 负责创建 goroutine，它也负责确保它可以停止 goroutine。

这个约定有助于确保你的程序在组合和扩展时可以扩展。我们将在本章后面的“channel”和“context 包”中重新讨论这种技术和规则。我们如何确保 goroutine 能够被停止，可以根据 goroutine 的类型和用途而有所不同，但是它们所有这些都是建立在完成 channel 传递的基础上的。

or-channel

有时你可能会发现自己希望将一个或多个完成的 channel 合并到一个完成的 channel 中，该 channel 在任何组件 channel 关闭时关闭。编写一个执行这种耦合的选择语句是完全可以接受的，尽管很冗长。但是，有时你无法知道你在运行时使用的已完成的 channel 的数量。在这种情况下，或者如果你只喜欢单线程，你可以使用 or-channel 模式将这些 channel 组合在一起。

这种模式通过递归和 goroutine 创建一个复合 done channel。我们来看一下：

```
var or func(channels ...<-chan interface{}) <-chan interface{}
or = func(channels ...<-chan interface{}) <-chan interface{} { ❶
    switch len(channels) {
    case 0: ❷
        return nil
    case 1: ❸
        return channels[0]
    }

    orDone := make(chan interface{})
    go func() { ❹
        defer close(orDone)

        switch len(channels) {
        case 2: ❺
             select {
             case <-channels[0]:
             case <-channels[1]:
             }
```

```
        default: ❻
            select {
            case <-channels[0]:
            case <-channels[1]:
            case <-channels[2]:
            case <-or(append(channels[3:], orDone)...): ❻
            }
        }
    }()
    return orDone
}
```

❶ 在这里，我们有我们的函数，或者，它采用可变的 channel 切片并返回单个 channel。

❷ 由于这是一个递归函数，我们必须设置终止标准。首先，如果可变切片是空的，我们只返回一个空 channel。这是由于不传递 channel 的观点所产生的，我们不希望复合的 channel 做任何事情。

❸ 我们的第二个终止标准是如果我们的变量切片只包含一个元素，我们只返回该元素。

❹ 这是函数的主体，以及递归发生的地方。我们创建了一个 goroutine，以便我们可以不受阻塞地等待我们 channel 上的消息。

❺ 基于我们进行迭代的方式，每一次迭代调用都将至少有两个 channel。在这里我们为需要两个 channel 的情况采用了约束 goroutine 数目的优化方法。

❻ 在这里，我们在循环到我们存放所有 channel 的 slice 的第三个索引的时候，我们创建了一个 or-channel 并从这个 channel 中选择了一个。这将形成一个由现有 slice 的剩余部分组成的树并且返回第一个信号量。为了使在建立这个树的 goroutine 退出的时候在树下的 goroutine 也可以跟着退出，我们将这个 orDone channel 也传递到了调用中。

这是一个相当简洁的函数，使你可以将任意数量的 channel 组合到单个 channel 中，只要任何组件 channel 关闭或写入，该 channel 就会关闭。我们来看看如何使用这个功能。下面是一个简短的例子，它将经过一段时间后关闭的 channel，并将这些 channel 合并到一个关闭的单个 channel 中：

```
sig := func(after time.Duration) <-chan interface{}{❶
    c := make(chan interface{})
    go func() {
        defer close(c)
        time.Sleep(after)
    }()
    return c
}
start := time.Now()❷
<-or(
       sig(2*time.Hour),
       sig(5*time.Minute),
       sig(1*time.Second),
       sig(1*time.Hour),
       sig(1*time.Minute),
)
fmt.Printf("done after %v", time.Since(start)) ❸
```

❶ 此功能只是创建一个 channel，当后续时间中指定的时间结束时将关闭该 channel。

❷ 在这里，我们大致追踪来自 or 函数的 channel 何时开始阻塞。

❸ 在这里，我们打印读取发生的时间。

如果你运行这个程序，你会得到：

```
done after 1.000216772s
```

请注意，尽管在我们的调用中放置了多个 channel 或需要不同时间才能关闭，但我们在 1s 后关闭的 channel 会导致由该调用创建的整个 channel 关闭。这是因为尽管它位于树或函数构建的树中，它将始终关闭，因此依赖于其关闭的 channel 也将关闭。

我们以附加的 goroutine 为代价来实现这个简洁性，$f(x)=\lfloor x/2 \rceil$，其中 x 是 goroutine 的数量，但要记住 Go 语言的一个优点是能够快速创建，调度和运行 goroutine，并且该语言积极鼓励使用 goroutine 来正确建模问题。担心在这里创建的 goroutine 的数量可能是一个不成熟的优化。此外，如果在编译时你不知道你正在使用多少个“done channel”，则将会没有其他方式可以合并“done channel”。

这种模式在你的系统中的模块交汇处非常有用。在这些交汇处，你的调用堆中应该有复数种的用来取消 goroutine 的决策树。使用 `or` 函数，你可以简单地将它们组合在一起并将其传递给堆栈。我们将在本章后面“context 包”中看到另一种做法，这也很好，也许更具描述性。

我们还将了解如何使用此模式的变体在第 5 章“复制请求”中形成更复杂的模式。

错误处理

在并发程序中，错误处理可能难以正确进行。有时候，我们花了很多时间思考我们的各种 stage 如何共享信息和进行协调，我们忘记考虑它们如何优雅地处理错误的状态。当 Go 语言避开了流行的错误异常模型时，它声明错误处理非常重要，并且在开发我们的程序时，我们应该给出我们的错误路径给予我们的算法同样的关注。本着这种精神，让我们来看看在处理多个并发进程时我们如何做到这一点。

思考错误处理时最根本的问题是，“谁应该负责处理错误？”在某些时候，程序需要停止将错误输出来，并且实际上对它做了些什么。这么做的目的是什么？

在并发进程中，这个问题变得更复杂一些。因为并发进程独立于其父进程或兄弟进程运行，所以它可能很难推断出错是正确的。查看下面的代码以查看此问题的示例：

```
checkStatus := func(
    done <-chan interface{},
    urls ...string,
) <-chan *http.Response {
    responses := make(chan *http.Response)
    go func() {
        defer close(responses)
        for _, url := range urls {
            resp, err := http.Get(url)
            if err!=nil {
```

```
                fmt.Println(err) ❶
                continue
            }
            select {
            case <-done:
                return
            case responses <- resp:
            }
        }
    }()
    return responses
}
done := make(chan interface{})
defer close(done)
urls := []string{"https://www.google.com", "https://badhost"}
for response := range checkStatus(done, urls...) {
        fmt.Printf("Response: %v\n", response.Status)
}
```

❶ 在这里，我们看到 goroutine 在尽最大努力表示出现错误。它还能做什么？它无法传回！有多少错误才是太多？它是否继续提出要求？

运行这个代码产生：

```
Response: 200 OK
Get https://badhost: dial tcp: lookup badhost on 127.0.1.1:53: no such host
```

在这里，我们看到在这个问题上 goroutine 没有选择。它不能简单地吞下错误，因此它只能做出明智的事情：它会打印错误并希望某些内容被关注。不要把你的 goroutine 放在这个尴尬的位置。我建议你分开你的顾虑：一般来说，你的并发进程应该把他们的错误发送到你的程序的另一部分，它有你的程序状态的完整信息，并可以做出更明智的决定做什么。以下示例演示了此问题的正确解决方案：

```
type Result struct {❶
       Error error
       Response *http.Response
}
checkStatus := func(done <-chan interface{}, urls ...string)<-chan Result {❷
    results := make(chan Result)
    go func() {
        defer close(results)
        for _, url := range urls {
             var result Result
             resp, err := http.Get(url)
             result = Result{Error: err, Response: resp}❸
             select {
```

```
                case <-done:
                        return
                case results <- result: ❹
                }
            }
        }()
        return results
    }
    done := make(chan interface{})
    defer close(done)
    urls := []string{"https://www.google.com", "https://badhost"}
    for result := range checkStatus(done, urls...) {
        if result.Error != nil { ❺
            fmt.Printf("error: %v", result.Error)
            continue
        }
        fmt.Printf("Response: %v\n", result.Response.Status)
    }
```

❶ 在这里，我们创建一个包含 * http.Response 和从我们的 goroutine 中的循环迭代中可能出现的错误的类型。

❷ 该行返回一个可读取的 channel，以检索循环迭代的结果。

❸ 在这里，我们创建一个 Result 实例，并设置错误和响应字段。

❹ 这是我们将结果写入我们的 channel 的地方。

❺ 在这里，在我们的 main goroutine 中，我们能够智能地处理由 checkStatus 启动的 goroutine 中出现的错误，以及更大程序的完整背景。

输出如下：

```
Response: 200 OK
error: Get https://badhost: dial tcp: lookup badhost on 127.0.1.1:53:
no such host
```

这里要注意的关键是我们如何将潜在的结果与潜在的错误结合起来。这表示从 goroutine checkStatus 创建的完整可能结果集，并且允许我们的主要常规关于发生错误时做什么的决定。从更广泛的角度来说，我们已经成功地将错误处理的担忧从我们的生产者 goroutine 中分离出来。这是可取的，因为生成

goroutine 的 goroutine（在这种情况下是我们的 main goroutine）具有更多关于正在运行的程序的上下文，并且可以做出关于如何处理错误的更明智的决定。

在前面的例子中，我们只是将错误写入 stdio（标准输入输出），但我们可以做其他的事情。让我们稍微修改我们的程序，以便在出现三个或更多错误时停止尝试检查状态：

```
done := make(chan interface{})
defer close(done)

errCount := 0
urls := []string{"a", "https://www.google.com", "b", "c", "d"}
for result := range checkStatus(done, urls...) {
    if result.Error != nil {
        fmt.Printf("error: %v\n", result.Error)
        errCount++
        if errCount >= 3 {
            fmt.Println("Too many errors, breaking!")
            break
        }
        continue
    }
    fmt.Printf("Response: %v\n", result.Response.Status)
}
```

代码产生如下输出：

```
error: Get a: unsupported protocol scheme ""
Response: 200 OK
error: Get b: unsupported protocol scheme ""
error: Get c: unsupported protocol scheme ""
Too many errors, breaking!
```

你可以看到，因为错误是从 checkStatus 返回的而不是在 goroutine 内部处理的，错误处理遵循熟悉的 Go 语言模式。这是一个简单的例子，但不难想象，main goroutine 正在协调多个 goroutine 的结果，并制定更复杂的规则来继续或取消子 goroutine。此外，这里的主要内容是，在构建从 goroutine 返回值时，应将错误视为一等公民。如果你的 goroutine 可能产生错误，那么这些错误应该与你的结果类型紧密结合，并且通过相同的通信线传递，就像常规的同步函数一样。

pipeline

当你编写一个程序时，你可能不会坐下来写一个长函数，至少我希望你不要！你以函数、结构体、方法等形式构造抽象。为什么要这样做？部分是为了抽象出与大流量无关的细节，另一部分是为了能够在不影响其他区域的情况下处理一个代码区域。你有没有必要对系统进行更改并发现你必须触及多个领域才能做出一个合乎逻辑的改变？这可能是因为该系统有糟糕的抽象。

pipeline 是你可以用来在系统中形成抽象的另一种工具。特别是，当你的程序需要流式处理或批处理数据时，它是一个非常强大的工具。pipeline 这个词据称是在 1856 年首次使用的，可能是指将液体从一个地方输送到另一个地方的一系列管道。我们在计算机科学中借用了这个术语，因为我们也在从一个地方向另一个地方传输某些东西：数据。pipeline 只不过是一系列将数据输入，执行操作并将结果数据传回的系统。我们称这些操作都是 pipeline 的一个 stage。

通过使用 pipeline，你可以分离每个 stage 的关注点，这提供了许多好处。你可以相互独立地修改各个 stage，你可以混合搭配 stage 的组合方式，而无需修改 stage，你可以将每个 stage 同时处理到上游或下游 stage，并且可以扇出或限制部分你的 pipeline。我们将在本章后面的“扇出，扇入”中介绍 fan-out，我们将在第 5 章介绍速率限制。你不必担心这些术语现在意味着什么，让我们从简单的开始，尝试构建一个 pipeline 的 stage。

如前所述，一个 stage 只是将数据输入，对其进行转换并将数据发回。这是一个可以被视为 pipeline stage 的函数：

```
multiply := func(values []int, multiplier int) []int {
    multipliedValues := make([]int, len(values))
    for i, v := range values {
        multipliedValues[i] = v * multiplier
    }
    return multipliedValues
}
```

这个函数用一个乘法器取一部分整数，随着它的增加循环遍历它们，并返回一个新的变换切片。看起来像一个无聊的函数，是吧？我们来创建另一个 stage：

```
add := func(values []int, additive int) []int {
    addedValues := make([]int, len(values))
    for i, v := range values {
        addedValues[i] = v + additive
    }
    return addedValues
}
```

另一个无聊透顶的函数！这个函数是创建一个新的切片，并为每个元素添加一个值。在这一点上，你可能想知道是什么使这两个函数成为了 pipeline 的 stage，而不仅仅是函数。让我们尝试将它们合并：

```
ints := []int{1, 2, 3, 4}
for _, v := range add(multiply(ints, 2), 1) {
    fmt.Println(v)
}
```

代码产生的输出：

```
3
5
7
9
```

看看我们如何在 `range` 子句中结合添加和乘法。这些函数就像你每天工作的函数一样，但是因为我们将它们构建为具有 pipeline stage 的属性，所以我们可以将它们组合起来形成一个 pipeline。那很有意思，pipeline stage 的属性是什么？

- 一个 stage 消耗并返回相同的类型。
- 一个 stage 必须用语言来表达，以便它可以被传递[注2]。Go 语言中的功能已被证实，并很好地适用于此目的。

注 2：在语言环境中，具体化意味着语言向开发人员展示了一个概念，以便他们可以直接使用它。据说 Go 语言中的函数是通用的，因为你可以定义具有函数签名类型的变量。这也意味着你可以在你的程序中传递函数。

那些熟悉函数式编程的人可能会点头，并思考像高阶函数和 *monad* 这样的术语。事实上，pipeline stage 与函数式编程密切相关，可以被认为是 monad 的一个子集。我不会在这里明确地讨论 monad 或函数式编程，但它们本身就是一个有趣的主题，并且在尝试理解 pipeline 时，对这两个主题的工作知识虽然不必要，但是有用。

在这里，我们的 `add` 和 `multiply` stage 满足 pipeline stage 的所有属性：它们都消耗一个 `int` 切片并返回一个 `int` 切片，并且因为 Go 语言具有函数化功能，所以我们可以传递 `add` 和 `multiply`。这些属性引起了我们前面提到的 pipeline stage 的有趣特性，即在不改变 stage 本身的情况下，将我们的 stage 结合到更高层次变得非常容易。

例如，如果我们现在想要为 pipeline 添加一个额外的 stage 来乘以 2，我们只需将我们以前的 pipeline 包装在一个新的乘法 stage，如下所示：

```
ints := []int{1, 2, 3, 4}
for _, v := range multiply(add(multiply(ints, 2), 1), 2) {
    fmt.Println(v)
}
```

运行这个代码将会产生如下结果：

```
6
10
14
18
```

注意我们如何在不编写新函数的前提下修改现有函数或修改我们 pipeline 的结果。也许你已经开始看到使用 pipeline 模式的好处了。当然，我们也可以在程序上编写这个代码：

```
ints := []int{1, 2, 3, 4}
for_,v:=rangeints{
        fmt.Println(2*(v*2+1))
}
```

最初，这看起来简单得多，但正如我们看到的那样，程序代码在处理数据流时不会提供与 pipeline 相同的好处。

请注意每个 stage 是如何获取切片数据并返回切片数据的？这些 stage 正在执行我们称作批处理的操作。这意味着它们仅对大块数据进行一次操作，而不是一次一个离散值。还有另一种类型的 pipeline stage 执行流处理。这意味着这个 stage 一次只接收和处理一个元素。

批处理和流处理有优点和缺点，我们将稍微讨论一下。现在，请注意，为使原始数据保持不变，每个 stage 都必须创建一个等长的新片段来存储其计算结果。这意味着我们程序在任何时候的内存占用量都是我们发送到我们 pipeline 开始处的片大小的两倍。让我们将我们的 stage 转换为以流为导向，看起来如下所示：

```
multiply := func(value, multiplier int) int {
    return value * multiplier
}
add := func(value, additive int) int {
    return value + additive
}
ints := []int{1, 2, 3, 4}
for _,v:=range ints{
    fmt.Println(multiply(add(multiply(v, 2), 1), 2))
}
```

代码产生如下结果：

```
6
10
14
18
```

每个 stage 都接收并发出一个离散值，我们的程序的内存占用空间将回落到只有 pipeline 输入的大小。但是我们不得不将 pipeline 写入到 `for` 循环的体内，并让 `range` 语句为我们的 pipeline 进行繁重的提升。这不仅限制了我们供应 pipeline 的重复使用，而且将在本章后面介绍，这也限制了我们的扩展能力。

还有其他问题。实际上，我们正在为循环的每次迭代实例化我们的 pipeline。尽管进行函数调用代价很低，但我们为循环的每次迭代进行三次函数调用。并发性又如何？我前面说过，使用 pipeline 的好处之一是能够同时处理各个 stage，并且我提到了一些关于扇出（fan-out）的内容。所有进来的地方在哪里？

我可能会扩展我们的乘法和增加一些功能来介绍这些概念，但已经完成了介绍流水线概念的工作。现在是时候开始学习在 Go 语言中构建 pipeline 的最佳实践了，它始于 Go 语言的 channel 基元。

构建 pipeline 的最佳实践

channel 非常适合在 Go 语言中构建 pipeline，因为它们满足了我们所有的基本要求。它们可以接受和产生值，可以安全地同时使用，还可以被放弃，它们被语言所证实。让我们花点时间转换一下前面的例子来改用 channel：

```
generator := func(done <-chan interface{}, integers ···int)<-chan int {
    intStream := make(chan int)
    go func() {
        defer close(intStream)
        for _, i := range integers {
            select {
            case <-done:
                    return
            case intStream <- i:
            }
        }
    }()
    return intStream
}

multiply := func(
done <-chan interface{},
intStream <-chan int,
multiplier int,
)<-chan int {
    multipliedStream := make(chan int)
    go func() {
        defer close(multipliedStream)
        for i := range intStream {
            select {
```

```
                case <-done:
                        return
                case multipliedStream <- i*multiplier:
                }
        }
    }()
    return multipliedStream
}

add := func(
done <-chan interface{},
      intStream <-chan int,
      additive int,
)<-chan int {
    addedStream := make(chan int)
    go func() {
        defer close(addedStream)
        for i := range intStream {
            select {
            case <-done:
                    return
            case addedStream <- i+additive:
            }
        }
    }()
    return addedStream
}

done := make(chan interface{})
defer close(done)

intStream := generator(done, 1, 2, 3, 4)
 pipeline := multiply(done, add(done, multiply(done, intStream, 2), 1), 2)

for v := range pipeline {
    fmt.Println(v)
}
```

这代码将产生如下输出：

```
6
10
14
18
```

看起来我们已经复制了期望的输出，但代价是更多的代码。我们究竟得到了什么？首先，我们来看看我们写的是什么。我们现在有三个函数，而不是两

个。它们都看起来像是在他们的函数体内开始了一个 goroutine，并使用了我们在本章前面“防止 goroutine 泄漏”中建立的模式，通过一个 channel 表示该 goroutine 应该退出。它们看起来都像是返回 channel，其中一些看起来像是在另外一个 channel 中。很有趣！让我们开始进一步分解：

```
done := make(chan interface{})
defer close(done)
```

我们的程序所做的第一件事是创建一个 done channel，并在 defer 语句中关闭它。正如前面所讨论的那样，这可以确保我们的程序干净地离开，不会泄漏 goroutine。没啥新东西。接下来，我们来看看函数 generator：

```
generator := func(done <-chan interface{}, integers ...int)<-chan int {
    intStream := make(chan int)
    go func() {
        defer close(intStream)
        for _, i := range integers {
            select {
            case <-done:
                return
            case intStream <- i:
            }
        }
    }()
    return intStream
}

// ...

intStream := generator(done, 1, 2, 3, 4)
```

generator 函数接受一个可变的整数切片，构造一个缓存长度等于输入整数片段的整数 channel，启动一个 goroutine 并返回构造的 channel。然后，在创建的 goroutine 上，generator 函数使用 range 语句遍历传入的可变切片，并在其创建的 channel 上发送切片的值。

请注意，channel 上的发送与完成 channel 上的选择共享一条 select 语句。再一次，这是我们在本章前面“防止 goroutine 泄漏”中建立的模式，以防止泄漏 goroutines。

简而言之，generator 函数将一组离散值转换为一个 channel 上的数据流。适当地说，这种类型的功能称为生成器。在使用流水线时，你会经常看到这一点，因为在流水线开始时，你总是会有一些需要转换为 channel 的数据。我们将稍微介绍一些有趣的生成器的例子，但我们先来完成对这个程序的分析。接下来，构建我们的 pipeline：

```
pipeline := multiply(done, add(done, multiply(done, intStream, 2), 1), 2)
```

这与我们一直在努力的流水线相同：对于一串数字，我们将它们乘以 2，加 1，然后将结果乘以 2。这个 channel 与我们前面例子中使用函数的 channel，但它在很重要的方面有所不同。

首先，我们正在使用 pipeline。这是显而易见的，因为它允许两件事情：在我们的 pipeline 的末尾，我们可以使用范围语句来提取值，并且在每个 stage 我们可以安全地同时执行，因为我们的输入和输出在并发上下文中是安全的。

这给我们带来了第二个不同之处：pipeline 的每个 stage 都在执行控制。这意味着任何 stage 只需要等待其输入，并且能够发送其输出。事实证明，这会产生巨大的影响，我们将在本章后面“扇出，扇入”中发现，但现在可以简单地注意到它允许我们的 stage 相互独立地执行某个片段时间。

最后，在我们的例子中，我们对这个 pipeline 进行了排序，并且通过系统获取了值：

```
for v := range pipeline {
    fmt.Println(v)
}
```

下面是一个表格，演示系统中的每个值如何进入每个 channel，以及 channel 何时关闭。迭代是我们所在的 `for` 循环迭代的基数为零的计数，每列的值是 pipeline stage 的值：

迭代	生产者	Multiply	Add	Multiply	值
0	1				
0		1			
0	2		2		
0		2		3	
0	3		4		6
1		3		5	
1	4		6		10
2	(closed)	4		7	
2		(closed)	8		14
3			(closed)	9	
3				(closed)	18

让我们更仔细地研究下我们使用这种模式是如何通知 goroutine 退出的。当我们处理多个相互依赖的 goroutine 时，这种模式如何最终起作用？如果我们在程序完成执行之前在完成的 channel 上调用 `close`，会发生什么情况？

为了回答这些问题，我们需要再来看看我们是如何创建 pipeline 的：

```
pipeline := multiply(done, add(done, multiply(done, intStream, 2), 1), 2)
```

这些 stage 通过两种方式相互关联：通过通用的 `done` channel，以及通往 pipeline 后续 stage 的 channel。换句话说，`multiply` 函数创建的 channel 被传递给 `add` 函数等。让我们重新审视前面的表格，并在允许完成之前，关闭完成的 channel，看看会发生什么：

迭代	生产者	Multiply	Add	Multiply	值
0	1				
0		1			
0	2		2		
0			2		3
1	3		4		6
close(done)	(closed)	3		5	
		(closed)	6		
			(closed)	7	
				(closed)	
					(exit range)

知道如何通过 pipeline 的传递来关闭 done channel 了吗？这是有 pipeline 的每个 stage 中的两件事来使之变为可能：

- 对传入的 channel 进行迭代，当传入的 channel 已经关闭了，range 语句也将会退出。
- 发送与完成 channel 共享 `select` 语句。

无论 pipeline stage 所处的是什么（在等待传入的 channel 的状态还是等待发送完成），关闭 done channel 都将会迫使 pipeline stage 被终止。

这里有一个复发关系。在 pipeline 开始时，我们已经确定我们必须将离散值转换为 channel。在这个过程中有两点必须是可抢占的：

- 创建几乎不是瞬时的离散值。
- 在其 channel 上发送离散值。

首先取决于你。在我们的例子中，在生成器函数中，离散值是通过遍历可变切片生成的，它足够瞬时，不需要被抢占。第二个是通过我们的 `select` 语句和 done channel 处理的，它确保发生器即使被阻塞试图写入 `intStream` 也是可抢占的。

在 pipeline 的另一端，通过感应确保最终 stage 的可抢占性。这是可抢占的，因为我们正在覆盖的 channel 在抢占时会被关闭，因此当这种情况发生时，我们的频道将会中断。最后 stage 是可抢占的，因为我们依赖的流是可抢占的。

在 pipeline 开始和结束之间，代码总是在使用 range 语句中遍历 channel，并且在一个包含一个 done channel 的 select 语句中向其他的 channel 发送消息。

如果某个 stage 在从传入 channel 检索到值时被阻止，则该 channel 关闭时它将变为未阻止状态。我们通过归纳知道渠道将被关闭，因为它或者是一个 stage，就像我们所处的 stage 一样，或者我们已经建立的 pipeline 的初始 stage 是可抢占的。如果某个 stage 在发送值时被阻塞，则由于 select 语句而可抢占。

因此，我们的整个 pipeline 始终可以通过关闭完成 channel 来抢占。很酷，对吧？

一些便利的生成器

我早些时候承诺我会谈论一些可能广泛使用的有趣的生成器。提醒一下，pipeline 的生成器是将一组离散值转换为 channel 上的值流的任何函数。我们来看看一个名为 repeat 的生成器：

```
repeat := func(
   done <-chan interface{},
   values ...interface{},
)<-chan interface{} {
    valueStream := make(chan interface{})
    go func() {
        defer close(valueStream)
        for {
             for _, v := range values {
                    select {
                    case <-done:
                             return
                    case valueStream <- v:
                    }
             }
        }
    }()
```

```
    return valueStream
}
```

这个函数会重复你传给它的值，直到你告诉它停止。让我们来看看另一个通用 pipeline stage，这个 pipeline stage 在重复使用时很有用，请参考：

```
take := func(
    done <-chan interface{},
    valueStream <-chan interface{},
    num int,
) <-chan interface{} {
    takeStream := make(chan interface{})
    go func() {
        defer close(takeStream)
        for i:=0;i<num;i++ {
            select {
            case <-done:
                return
            case takeStream <- valueStream:
            }
        }
    }()
    return takeStream
}
```

这个 pipeline stage 只会从其传入的 valueStream 中取出第一个 num 项目，然后退出。两者结合在一起可以非常强大：

```
done := make(chan interface{})
defer close(done)

for num := range take(done, repeat(done, 1), 10) {
      fmt.Printf("%v ", num)
}
```

运行这个代码将产生：

```
1 1 1 1 1 1 1 1 1 1
```

在这个基本的例子中，我们创建了一个重复生成器来生成无限数量的数字 1，但是只取前 10 个。由于重复生成器的发送块在接收 stage 的接收，重复生成器非常有效。尽管我们有能力生成无限数量的流，但我们只生成 N + 1 个实例，其中 N 是我们传递到 take stage 的数量。

我们可以扩展这一点。让我们创建另一个重复的生成器，但是这次我们创建一个重复调用函数的生成器。我们称之为 repeatFn：

```
repeatFn := func(
    done <-chan interface{},
    fn func() interface{},
) <-chan interface{} {
    valueStream := make(chan interface{})
    go func() {
        defer close(valueStream)
        for {
            select {
            case <-done:
                return
            case valueStream <- fn():
            }
        }
    }()
    return valueStream
}
```

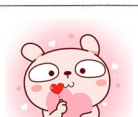
我们用它来生成 10 个随机数字：

```
done := make(chan interface{})
defer close(done)

rand := func() interface{} { return rand.Int()}

for num := range take(done, repeatFn(done, rand), 10) {
    fmt.Println(num)
}
```

输出如下：

```
5577006791947779410
8674665223082153551
6129484611666145821
4037200794235010051
3916589616287113937
6334824724549167320
605394647632969758
1443635317331776148
894385949183117216
2775422040480279449
```

这非常酷，一个根据需要生成随机整数的无限 channel！



你可能想知道为什么所有这些发生器和 stage 都在 `interface{}` 的 channel 上接收和发送数据。我们可以很容易地将这些函数写成特定的类型，或者可以编写 Go 语言生成器。

Go 语言中的空接口有点不太好，但对于 pipeline stage，我认为可以处理 `interface {}` 的 channel，以便使用标准的 pipeline 模式库。正如我们前面所讨论的，许多 pipeline 的效用来自可重用的 stage。当 stage 以适合自身的特异性水平进行操作时，这是最好的。在 `repeat` 和 `repeatFn` 生成器中，关心的是通过在列表或运算符上循环来生成数据流。随着 `take` stage，担忧正在限制我们的 pipeline。这些操作都不需要关于它们正在处理的类型的信息，而只需要知道参数的类型。

当你需要处理特定的类型时，你可以放置一个为你执行类型断言的 stage。有一个额外的 pipeline stage（因此 goroutine）和类型断言的性能开销可以忽略不计，正如我们稍后会看到的。下面是一个介绍 `toString` pipeline stage 的小例子：

```
toString := func(
    done <-chan interface{},
    valueStream <-chan interface{},
) <-chan string {
    stringStream := make(chan string)
    go func() {
        defer close(stringStream)
        for v := range valueStream {
            select {
            case <-done:
                return
            case stringStream <- v.(string):
            }
        }
    }()
    return stringStream
}
```

以及如何使用它的一个例子：

```
done := make(chan interface{})
defer close(done)
```


```
var message string
for token := range toString(done, take(done, repeat(done, "I", "am."), 5)) {
    message += token
}

fmt.Printf("message: %s...", message)
```

输出如下：

```
message: Iam.Iam.I…
```

因此，让我们向自己证明，将通用部分 pipeline 的性能成本忽略不计。我们将编写两个基准测试函数：一个测试通用 stage，一个测试类型特定 stage：

```
func BenchmarkGeneric(b *testing.B) {
    done := make(chan interface{})
    defer close(done)

    b.ResetTimer()
    for range toString(done, take(done, repeat(done, "a"), b.N)) {
    }
}

func BenchmarkTyped(b *testing.B) {
    repeat := func(done <-chan interface{}, values ...string) <-chan string {
        valueStream := make(chan string)
        go func() {
            defer close(valueStream)
            for {
                for _, v := range values {
                    select {
                    case <-done:
                        return
                    case valueStream <- v:
                    }
                }
            }
        }()
        return valueStream
    }

    take := func(
        done <-chan interface{},
        valueStream <-chan string,
        num int,
    ) <-chan string {
        takeStream := make(chan string)
        go func() {
            defer close(takeStream)
```


```
            for i := num; i > 0 || i == -1; {
                if i != -1 {
                    i--
                }
                select {
                case <-done:
                    return
                case takeStream <- <-valueStream:
                }
            }
        }()
        return takeStream
    }

    done := make(chan interface{})
    defer close(done)

    b.ResetTimer()
    for range take(done, repeat(done, "a"), b.N) {
    }
}
```

运行这段代码的结果是：

```
BenchmarkGeneric-4              1000000                 2266    ns/op
BenchmarkTyped-4                1000000                 1181    ns/op
PASS
ok                              command-line-arguments          3.486s
```

你可以看到，类型相关的 stage 运行速度是非类型相关的两倍，但速度只有极快。一般来说，pipeline 上的限制因素将是你的生成器，或者是计算密集型的一个 stage。如果生成器不像 `repeat` 和 `repeatFn` 生成器那样从内存中创建流，则可能会受I/O限制。从磁盘或网络中读取数据可能会使测试结果显示在此处。

如果你的一个 stage 在计算上花费很大，那么这肯定会使这种性能开销消失。如果这种技术在你的嘴里仍然留下不好的味道，你总是可以编写一个 Go 语言生成器来创建你的发生器 stage。说到一个 stage 计算成本很高，我们该如何帮助缓解这个问题呢？它不会限制整个 pipeline 的速度吗？

为了缓解这种情况，让我们来讨论扇出扇入（fan-out, fan-in）技术。

扇入，扇出

假设你已经建立了一条 pipeline。数据流畅地流过你的系统，在你连接在一起的各个 stage 进行转换。它就像一条美丽的溪流（stream）；一个美丽的，缓慢的溪流，哦，上帝啊！为什么这需要这么久？

有时候，pipeline 中的各个 stage 可能在计算上特别昂贵。发生这种情况时，你的 pipeline 中的上游 stage 可能会被阻塞，同时等待昂贵的 stage 来完成。不仅如此，pipeline 本身可能需要很长时间才能全部执行。我们如何解决这个问题？

pipeline 的一个有趣属性是它们能够让你使用独立的，并且可以常常重新排序的 stage 的组合来操作数据流。你甚至可以多次重复使用 pipeline 的各个 stage。在多个 goroutine 上重用我们的 pipeline 的单个 stage 以试图并行化来自上游 stage 的 pull 是不是很有趣？也许这将有助于提高 pipeline 的性能。

事实上，事实证明它可以，而这种模式有一个名字：扇入，扇出。

扇出是一个术语，用于描述启动多个 goroutines 以处理来自 pipeline 的输入的过程，并且扇入是描述将多个结果组合到一个 channel 的过程中的术语。

那么什么时候一个 pipeline 的 stage 适合利用这种模式呢？如果以下两种情况适用，你可以考虑在某个 stage 使用：

- 它不依赖于之前 stage 计算的值。
- 运行需要很长时间。

循序独立性很重要，因为你无法保证你的 stage 的并发副本以何种顺序运行，也无法保证其返回的顺序。

我们来看一个例子。在下面的例子中，我构建了一个用来找到素数的非常低效的函数。我们将使用在本章前面“pipeline”中创建的许多 stage：

```
rand := func() interface{} { return rand.Intn(50000000) }

done := make(chan interface{})
defer close(done)

start := time.Now()

randIntStream := toInt(done, repeatFn(done, rand))
fmt.Println("Primes:")
for prime := range take(done, primeFinder(done, randIntStream), 10) {
        fmt.Printf("\t%d\n", prime)
}

fmt.Printf("Search took: %v", time.Since(start))
```

这些代码的输出结果如下：

```
Primes:
        24941317
        36122539
        6410693
        10128161
        25511527
        2107939
        14004383
        7190363
        45931967
        2393161
    Search took: 23.437511647s
```

我们生成一串随机数，最高为 50000000，将数据流转换为整数流，然后将其传入我们的 `primeFinder` stage。`primeFinder` 天真地开始试图将输入流上的数字除以它的每个数字。如果不成功，它会将该值传递到下一个 stage。当然，这是尝试找到素数的可怕方法，但它符合我们pipeline将花费很长时间的要求。

在我们的循环中，我们搜索找到的素数，在它们进入时将其打印出来，并感谢我们的 stage，在找到 10 个素数后关闭 pipeline。然后，我们打印出搜索花了多长时间，并通过 `defer` 语句关闭了完成的 channel，并且将 pipeline 关闭。

为了避免重复结果出现在我们的结果集中，我们可以在 pipeline 中引入另一个 stage 来缓存已在集合中找到的素数，但为了简单起见，我们将忽略这些。

你可以看到大概需要 23s 才能找到 10 个素数。不是很好。通常我们首先看一下算法本身，也许拿出一本算法手册，看看我们是否可以在每个 stage 进行改进。但是，由于这里的 stage 本身的目的就是缓慢的，我们将看看我们如何能够展开一个或多个 stage，以便更快地熬过慢速操作。

这是一个相对简单的例子，而我们只有两个 stage：随机数生成和素数筛选。在一个更大的程序中，你的 pipeline 可能由更多的 stage 组成，我们怎么知道哪一个扇出？请记住我们之前的标准：独立性和持续时间。我们的随机整数发生器肯定是顺序无关的，但运行起来并不需要很长时间。`primefinder` stage 也是顺序无关的，数字不是主要的，因为我们的朴素算法，它肯定需要很长时间才能运行。它看起来是一个很好的候选。

幸运的是，在 pipeline 中分散 stage 的过程非常容易。我们所要做的就是启动该 stage 的多个版本。所以不是这样的：

```
primeStream := primeFinder(done, randIntStream)
```

我们可以做这样的事情：

```
numFinders := runtime.NumCPU()
finders := make([]<-chan int, numFinders)
 for i := 0; i < numFinders; i++ {
        finders[i] = primeFinder(done, randIntStream)
}
```

在这里我们启动了这个 stage 的许多副本，因为我们有多个 CPU 核心。在我的计算机上，`runtime.NumCPU()` 返回 8，所以我们将继续在我们的讨论中使用这个数字。在生产中，我们可能会做一些经验性的测试来确定 CPU 的最佳数量，但在这里我们将保持简单，并且假设只有一个 `findPrimes` stage 的 CPU 会被占用。

就是这样！我们现在有 8 个 goroutine 从随机数发生器中拉出并试图确定数字是否为素数。生成随机数不应该花费太多时间，因此 `findPrimes` stage 的每个 goroutine 应该能够确定它的数字是否为素数，然后立即有另一个随机数可用。

但是我们仍然有一个问题：既然我们有 4 个 goroutine，还有 4 个 channel，但是我们的素数范围仅等于一个 channel。这将是我们使用扇入（fan-in）模式的绝佳实例。

正如我们前面所讨论的，扇入意味着将多个数据流复用或合并成一个流。这样做的算法相对简单：

```
fanIn := func(
    done <-chan interface{},
    channels ...<-chan interface{},
) <-chan interface{} { ❶
    var wg sync.WaitGroup ❷
    multiplexedStream := make(chan interface{})

    multiplex := func(c <-chan interface{}) { ❸
        defer wg.Done()
        for i := range c {
            select {
            case <-done:
                return
            case multiplexedStream <- i:
            }
        }
    }

    // 从所有的 channel 里取值
    wg.Add(len(channels)) ❹
    for _, c := range channels {
        go multiplex(c)
    }

    // 等待所有的读操作结束
    go func() { ❺
        wg.Wait()
        close(multiplexedStream)
    }()

    return multiplexedStream
}
```

❶ 在这里，我们采用我们的标准 done channel 来允许我们的 goroutine 被拆除，然后是一个可变的 interface{} channel 用来扇入。

❷ 在这一行中，我们创建了一个 sync.WaitGroup，以便我们可以等待知道所有 channel 都已耗尽。

❸ 在这里，我们创建一个函数 multiplex，它在传递时将从 channel 中读取，并将读取的值传递到 multiplexedStream channel。

❹ 此行将 sync.WaitGroup 增加我们多重 channel 的数量。

❺ 在这里，我们创建一个 goroutine 来等待我们多路复用的所有 channel 被耗尽，这样我们可以关闭 multiplexedStream channel。

简而言之，扇入涉及创建用户将读取的多路复用 channel，然后为每个传入 channel 启动一个 goroutine，以及在传入 channel 全部关闭时关闭复用 channel 的 goroutine。由于我们要创建一个等待 N 个其他分区完成的 goroutine，创建一个 sync.WaitGroup 来协调是很有意义的。多路复用功能还通知 WaitGroup 它已完成。

额外的提醒

如果结果到达的顺序不重要，那么对于扇入，扇出算法的幼稚实现将起作用。我们没有做任何事情来保证从 randIntStream 中读取项目的顺序在筛选过程中保留下来。稍后，我们将看一个保持运行顺序的例子。

让我们把所有这些放在一起，看看我们在运行时是否有所减少：

```
done := make(chan interface{})
defer close(done)

start := time.Now()

rand := func() interface{} { return rand.Intn(50000000) }

randIntStream := toInt(done, repeatFn(done, rand))

numFinders := runtime.NumCPU()
fmt.Printf("Spinning up %d prime finders.\n", numFinders)
finders := make([]<-chan interface{}, numFinders)
fmt.Println("Primes:")
for i := 0; i < numFinders; i++ {
    finders[i] = primeFinder(done, randIntStream)
}
```


```
for prime := range take(done, fanIn(done, finders...), 10) {
    fmt.Printf("\t%d\n", prime)
}

fmt.Printf("Search took: %v", time.Since(start))
```

输出结果如下：

```
Spinning up 8 prime finders.
Primes:
    6410693
    24941317
    10128161
    36122539
    25511527
    2107939
    14004383
    7190363
    2393161
    45931967
Search took: 5.438491216s
```

所以从大概 23 秒降到大概 5 秒，还不错！这清楚地表明了扇入，扇出模式的好处，它重新定义了管道的用途。我们将执行时间缩短了大约 78%，也不会大幅改变程序的结构。

or-done-channel

有时候，你需要处理来自系统各个分散部分的 channel。与 pipeline 所不同的是，你不能对一个被 done channel 所取消的 channel 将会进行什么行为做任何的断言。也就是说，你不知道你的 goroutine 是否被取消，这意味着你正在读取的 channel 将被取消。出于这个原因，正如在本章前面“防止 goroutine 泄漏”中所阐述的那样，我们需要用 channel 中的 select 语句来包装我们的读操作，并从已完成的 channel 中进行选择。这非常好，但是这样做需要的代码很容易读取，如下所示：

```
for val := range myChan {
   // 用 val 执行某些操作

}
```

如下操作可能会使 goroutine 激增：

```
loop:
for {
    select {
    case <-done:
        break loop
    case maybeVal, ok := <-myChan:
        if ok == false {
            return // 或许从 for 循环中退出
        }
       // 用 val 执行某些操作
    }
}
```

这可能会很快就繁忙起来，特别是如果你有嵌套循环。继续使用 goroutine 编写更清晰的并发代码，而不是过早优化，我们可以用一个 goroutine 来解决这个问题。我们封装了详细信息，以便于其他的函数：

```
orDone := func(done, c <-chan interface{}) <-chan interface{} {
    valStream := make(chan interface{})
    go func() {
        defer close(valStream)
        for {
            select {
            case <-done:
                return
            case v, ok := <-c:
                if ok == false {
                    return
                }
                select {
                case valStream <- v:
                case <-done:
                }
            }
        }
    }()
    return valStream
}
```

这样做可以让我们回到简单的循环，就像这样：

```
for val := range orDone(done, myChan) {
    // 用 val 执行某些操作
}
```

你可能会在你的代码中发现需要使用一系列 select 语句的紧密循环的边界案例，但我会鼓励你先尝试编写具有可读性的代码，并避免过早优化。

tee-channel

有时候你可能想分割一个来自 channel 的值，以便将它们发送到你的代码的两个独立区域中。设想一下，一个传递用户指令的 channel：你可能想要在一个 channel 上接收一系列用户指令，将它们发送给相应的执行器，并将它们发送给记录命令以供日后审计的东西。

从类 UNIX 系统中的 tee 命令中获得它的名字，*tee-channel* 就是这样做的。你可以将它传递给一个读 channel，并且它会返回两个单独的 channel，以获得相同的值：

```
tee := func(
    done <-chan interface{},
    in <-chan interface{},
) (_, _ <-chan interface{}) { <-chan interface{}) {
    out1 := make(chan interface{})
    out2 := make(chan interface{})
    go func() {
        defer close(out1)
        defer close(out2)
        for val := range orDone(done, in) {
            var out1, out2 = out1, out2 ❶
            for i := 0; i < 2; i++ { ❷
                select {
                case <-done:
                case out1<-val:
                    out1 = nil  ❸
                case out2<-val:
                    out2 = nil  ❸
                }
            }
```

```
        }
    }()
    return out1, out2
}
```

❶ 我们将要使用 out1 和 out2 的本地版本，所以我们会隐藏这些变量。

❷ 我们将使用一条 select 语句，以便不阻塞的写入 out1 和 out2。为确保两者都写入，我们将执行 select 语句的两次迭代：每个出站一个 channel。

❸ 一旦我们写入了 channel，我们将其影副本设置为 nil，以便进一步阻塞写入，而另一个 channel 可以继续。

注意写入 out1 和 out2 是紧密耦合的。直到 out1 和 out2 都被写入，迭代才能继续。通常来说这并不是一个问题，因为处理来自每个 channel 的吞吐量都应该是一个确定的某个之外而不是像 tee 命令那样，但这并没有任何的价值。下面是一个快速示例：

```
done := make(chan interface{})
defer close(done)

out1, out2 := tee(done, take(done, repeat(done, 1, 2), 4))

for val1 := range out1 {
    fmt.Printf("out1: %v, out2: %v\n", val1, <-out2)
}
```

通过这种模式，对于你的系统来说，继续使用 channel 作为“join 点”将会是易如反掌的事。

桥接 channel 模式

在某些情况下，你可能会发现自己希望从一系列的 channel 中消费产生的值：

```
<-chan <-chan interface{}
```

这与将 channel 切片合并到单个 channel 中稍有不同，如我们在本章前面“The or-channel”或“扇出，扇入”中所看到的。一系列的 channel 需要有序地写入，

即使是不同的来源。其中一个例子是一个整个生命周期都是间歇的 pipeline stage。如果按照我们在本章前面“约束”中建立的方式并确保 channel 都被写入它们的 goroutine 拥有，在每一个新的 goroutine 中的 pipeline stage 重启的时候，都会创建一个新的 channel。这也就意味着我们很有效地拥有了一个 channel 系列。我们将在第 5 章“治愈不健康的 goroutine”中详细探讨这种情况。

作为消费者，代码可能不关心其值来自于一系列的 channel 的事实。在这种情况下，处理一个充满 channel 的 channel 的可能会很多。如果我们定义一个功能，可以将充满 channel 的 channel 拆解为一个简单的 channel（一种称为桥接 channel 的技术），这将使消费者更容易关注手头的问题。以下是我们如何实现这一目标的一个例子：

```
bridge := func(
    done <-chan interface{},
    chanStream <-chan <-chan interface{},
) <-chan interface{} {
    valStream := make(chan interface{})❶
    go func() {
        defer close(valStream)
        for {❷
            var stream <-chan interface{}
            select {
            case maybeStream, ok := <-chanStream:
                if ok==false {
                    return
                }
                stream = maybeStream
            case <-done:
                return
            }
            for val := range orDone(done, stream) {❸
                select {
                case valStream <- val:
                case <-done:
                }
            }
        }
    }()
    return valStream
}
```

❶ 这是将返回 bridge 中的所有值的 channel。

❷ 这个循环负责从 chanStream 中提取 channel 并将其提供给嵌套循环来使用。

❸ 该循环负责读取已经给出的 channel 中的值，并将这些值重复到 valStream 中。当我们当前正在循环的流关闭时，我们从执行从此 channel 读取的循环中跳出，并继续循环的下一次迭代，选择要读取的 channel。这为我们提供了一个不间断的结果值的流。

这是非常简单的代码。现在我们可以使用桥接来实现一个在一个包含多个 channel 的 channel 上实现一个单 channel 的门面（facade）。下面是一个例子，它创建了 10 个 channel，每个 channel 都写入一个元素，并将这些 channel 传递给桥接函数：

```
genVals := func() <-chan <-chan interface{} {
    chanStream := make(chan (<-chan interface{}))
    go func() {
        defer close(chanStream)
        for i:=0;i<10;i++ {
            stream := make(chan interface{}, 1)
            stream <- i
            close(stream)
            chanStream <- stream
        }
    }()
    return chanStream
}
for v := range bridge(nil, genVals()) {
    fmt.Printf("%v ", v)
}
```

运行这将产生：

```
0 1 2 3 4 5 6 7 8 9
```

通过桥接，我们可以在单个 range 语句中使用处理 channel 的 channel，并专注于我们的循环逻辑。解构处理 channel 的 channel 留给特定于这个问题的代码。

队列排队

有时，在你的队列尚未准备好的时候就开始接受请求是很有用的。这个过程被称作队列。

这也就意味着只要你的 stage 完成了某些工作，它就会把结果存储在一个稍后其他 stage 可以获取到结果的临时存储位置，而且你的 stage 不需要保存一份指向结果的引用。在第 3 章“channel”中，我们讨论了带缓存的 channel，那其实就是一种队列，而且我们当时有足够的理由不去过多讨论使用它。

虽然在系统中引入队列功能非常有用，但它通常是优化程序时希望采用的最后一种技术之一。预先添加队列可以隐藏同步问题，例如死锁和活锁，并且，随着程序向正确性收敛，你可能会发现需要更多或更少的队列。

那么队列有什么好处呢？让我们开始回答这个问题，通过解决人们在调整系统性能时犯的一个常见错误：引入队列来尝试解决性能问题。队列几乎不会加速程序的总运行时间，它只会让程序的行为有所不同。

为了理解它的原因，我们来看看一个简单的 pipeline：

```
done := make(chan interface{})
defer close(done)

zeros := take(done, 3, repeat(done, 0))
short := sleep(done, 1*time.Second, zeros)
long := sleep(done, 4*time.Second, short)
pipeline := long
```

该管道将 4 个 stage 连在一起：

1. 一个重复的 stage，会产生层出不穷的 0。
2. 当 pipeline 中有 3 个元素的时候会取消上一个 stage，即取消产生“无限个 0”的 stage。

3. 暂停 1s 的“短”stage。

4. 一个长的，暂停 4s 的 stage。

对于这个例子来说，我们假设 stage 1 和 stage 2 是即时完成的，让我们关注休眠 stage 如何影响 pipeline 的运行时间。

下面的这个表格是用来展示时间（t），迭代变量（i），以及“长”“短”stage 运行到取下一个值所需要的时间。

Time(t)	i	Long stage	Short stage
0	0		1s
1	0	4s	1s
2	0	3s	(blocked)
3	0	2s	(blocked)
4	0	1s	(blocked)
5	1	4s	1s
6	1	3s	(blocked)
7	1	2s	(blocked)
8	1	1s	(blocked)
9	2	4s	(close)
10	2	3s	
11	2	2s	
12	2	1s	
13	3	(close)	

你可以看到这个 pipeline 需要大约 13s 的时间来运行。短暂的 stage 大约需要 9s 才能完成。

如果我们修改管道以包含缓冲区会发生什么？让我们来看看在长期和短期之间引入 2 的缓冲区的相同管道：

```
done := make(chan interface{})
defer close(done)
```

```
zeros := take(done, 3, repeat(done, 0))
short := sleep(done, 1*time.Second, zeros)
buffer := buffer(done, 2, short) // Buffers sends from short by 2
long := sleep(done, 4*time.Second, short)
pipeline := long
```

这是运行时间：

Time(t)	i	Long stage	Buffer	Short stage
0	0		0/2	1s
1	0	4s	0/2	1s
2	0	3s	1/2	1s
3	0	2s	2/2	(close)
4	0	1s	2/2	
5	1	4s	1/2	
6	1	3s	1/2	
7	1	2s	1/2	
8	1	1s	1/2	
9	2	4s	0/2	
10	2	3s	0/2	
11	2	2s	0/2	
12	2	1s	0/2	
13	3	(close)		

整个管道仍然需要 13s！但看看短期的运行时间。它仅在 3s 后完成，而不是之前的 9s。我们已经将这个 stage 的运行时间减少了三分之二！但是如果整个管道仍然需要 13s 来执行，这对我们有什么帮助？

让我们来看下面这个 pipeline：

```
p := processRequest(done, acceptConnection(done, httpHandler))
```

这里 pipeline 在取消之前不会退出，并且接收连接的 stage 不会停止接收连接，直到取消 channel。在这种情况下，你不希望到你程序的链接超时，因为你的 `processRequest` stage 阻止了你的 `acceptConnection` stage。你希望尽可能地

解除你的 acceptConnection stage。否则，你的程序的用户可能会开始发现他们的请求完全被拒绝。

因此，对于引入队列的效用问题的答案并不是一个 stage 的运行时间已经减少，而是它处于阻塞状态的时间减少了。这可以让这个 stage 继续工作。在这个例子中，用户可能会在他们的请求中经历滞后，但他们不会被拒绝服务。

通过这种方式，队列的真正用途是将 stage 分离，以便一个 stage 的运行时间不会影响另一个 stage 的运行时间。以这种方式解耦 stage，然后级联以改变整个系统的运行时行为，这取决于你的系统，可以是好的也可以是不好的。

然后我们来讨论调整你的排队问题。队列应该放在哪里？缓冲区大小应该是多少？这些问题的答案取决于你的管道的性质。

首先分析排队可以提高系统整体性能的情况。唯一适用的情况是：

- 如果在一个 stage 批处理请求节省时间。
- 如果 stage 中的延迟产生反馈回路进入系统。第一种情况的一个例子是将输入缓冲到比被设计为发送给（例如，盘）更快的事物（例如，存储器）的 stage。

很显然，这就是 Go 语言的 bufio 包的目的。下面是一个示例，演示了缓冲写入队列与未缓冲写入的简单比较：

```
func BenchmarkUnbufferedWrite(b *testing.B) {
    performWrite(b, tmpFileOrFatal())
}

func BenchmarkBufferedWrite(b *testing.B) {
    bufferredFile := bufio.NewWriter(tmpFileOrFatal())
    performWrite(b, bufio.NewWriter(bufferredFile))
}

func tmpFileOrFatal() *os.File {
    file, err := ioutil.TempFile("", "tmp")
```

```
    iferr!=nil {
        log.Fatal("error: %v", err)
    }
    return file
}

func performWrite(b *testing.B, writer io.Writer) {
    done := make(chan interface{})
    defer close(done)

    b.ResetTimer()
    for bt := range take(done, repeat(done, byte(0)), b.N) {
        writer.Write([]byte{bt.(byte)})
    }
}

go test -bench=. src/concurrency-patterns-in-go/queuing/buffering_test.go
```

以下是运行此基准的结果：

```
BenchmarkUnbufferedWrite-8    500000          3969          ns/op
BenchmarkBufferedWrite-8      1000000         1356          ns/op
PASS
ok                            command-line-arguments 3.398s
```

如预期的那样，缓冲写入比未缓冲写入更快。这是因为在 `bufio.Writer` 中，写入在内部排队到缓冲区中，直到已经积累了足够的块为止，然后块被写出。这个过程通常称为分块，原因很明显。

分块速度更快，因为 `bytes.Buffer` 必须增加其分配的内存以容纳它必须存储的字节。出于各种原因，增长的内存消耗是昂贵的。所以，我们需要增长的时间越少，整个系统的整体效率就越高。因此，排队提高了整个系统的性能。

这只是一个简单的内存分块示例，但是你可能会在该领域频繁地进行分块。通常，任何时候执行操作都需要开销，分块可能会提高系统性能。这方面的一些例子是打开数据库事务，计算消息校验和以及分配连续空间。

除了分块之外，如果你的算法可以通过支持向后看或排序进行优化，排队也可以起到帮助作用。

第二种情况是，一个 stage 的延迟导致管道中接收到了更多的输入，这更难以发现，但也更重要，因为它可能导致上游系统的崩溃。

这个想法通常被称为负反馈循环，向下螺旋，甚至是死亡螺旋。这是因为管道与上游系统之间存在经常性关系，上游 stage 或系统提交新请求的速度在某种程度上与管道的有效性有关。

如果管道的效率降低到某个临界阈值以下，管道上游的系统开始增加它们对管道的输入，这导致管道损失更多效率，并且死亡螺旋开始。如果没有某种安全防护，使用管道的系统将永远不能恢复。

通过在管道入口处引入队列，你可以用创建请求滞后为代价来打破反馈循环。从调用者进入管道的角度来看，请求似乎正在处理中，但需要很长时间。只要调用者不超时，你的管道将保持稳定。如果主叫方超时，则需要确保你在出列时支持某种检查准备情况。如果你不这样做，你可能会无意中通过处理死亡请求来创建反馈循环，从而降低管道的效率。

你有没有见过死亡螺旋？

如果你曾尝试访问一些热门的新系统（例如，新游戏服务器，用于产品发布的网站等），并且尽管开发人员尽了最大的努力，但该网站一直处于弹跳状态，恭喜！你可能目睹了一个负面反馈的循环。

开发团队开始尝试不同的修复方法，直到有人意识到他们需要一个队列，并且开始匆忙实施。

然后客户开始抱怨排队时间！

所以从我们的例子中，我们可以看到一种模式的出现，需要实现排队模式：

- 在你的管道入口处。

- 在这个 stage，批量操作将会带来更高的效率。

你可能会尝试在其他的地方增加队列。例如，在一个重度计算 stage 之后。但是，请避免那样的尝试！就像我们所学到的，只有在很少的情况下队列可以减少你管道的运行时间。而且，在队列中胡乱操作可能会导致灾难性的后果。或许，在最初这并不会显得很明显，我们需要讨论管道的吞吐量来了解为什么会导致这个情况的原因。别担心，这并不难。而且他会帮我们回答对于如何决定需要用多大的队列的问题。

在“队列”理论中，有这样的一条法则，通过足够的取样，可以预测管道的需求率。这被称作利特尔法则。我们首先定义利特尔法则的共识。它一般被表达为：$L=\lambda W$，其中：

- L= 系统中平均负载数。
- λ = 负载的平均到达率。
- W = 负载在系统中花费的平均时间。

这个等式只可以应用在所谓的稳定的系统。在管道中，一个稳定的系统是指工作负载进入管道，或者说入口的速率与负载退出系统的速率相等。如果入口的速率超过了出口的速率，你的系统就是不稳定的，而且已经进入了一个死循环。如果你的入口速率没有超过出口速率，你仍旧造成了一个不稳定的系统，但这所导致的也仅仅是你的系统资源并没有被完全使用而已。那并不是这个世界上最糟糕的情况，但是，你可能会在大规模系统（例如，集群或者数据中心）中关心这个问题。

所以，让我们先认为我们的管道是稳定的，如果想要将 W（也就是一个负载）在我们系统中所花费的平均时间减少一个系数 n。而且我们在增加系统出口的速率的时候，只能减少系统中存在的平均负载的数量。需要注意的是，如果在我们的 stage 中增加了队列，我们其实是在增加 L，也就是说，或是增加了

负载到达的速率（nL = nλ * W），或是增加了一个负载在系统中花费的平均时间（nL = λ * nW）。通过利特尔法则，我们已经证明了队列不会有助于减少在系统中所花费的时间。

同时请注意，由于我们正在观察整个管道，因此将 W 减少 n 倍将分布在我们管道的所有 stage。在我们的案例中，小法的定义应该是这样的：

$$L = \lambda \Sigma_i W_i$$

这是另一种说法，你的管道只会和最慢的 stage 一样快。不分青红皂白地优化！

所以利特尔法则很整洁！这个简单的等式打开了各种分析我们管道的方法。让我们用它来问一些有趣的问题。在我们的分析中，假设我们的管道有三个 stage。

让我们尝试确定我们的管道每秒可以处理多少个请求。假设我们在我们的流水线上启用采样，并发现 1 个请求（r）需要大约 1s 才能通过流水线。让我们插入这些数字！

```
3r = λr/s * 1s

3r/s = λr/s

λr/s = 3r/s
```

我们将 L 设置为 3，因为我们管道中的每个 stage 都在处理请求。然后我们将 W 设置为 1s，做一个小代数，然后瞧！在这个管道中，我们每秒可以处理三个请求。

如何确定我们的队列需要多大才能处理所需数量的请求。利特尔法则能帮助我们回答这个问题吗？

假设我们的采样表明请求需要 1 ms 来处理。我们的队列需要处理每秒 100000 次请求的大小是多少？让我们再一次插入数字！

```
Lr-3r = 100000r/s × 0.0001s

Lr-3r = 10r

Lr = 7r
```

同样，我们的 pipeline 有三个 stage，所以我们将 L 递减 3，λ 设置为 100000r/s，并且发现如果我们想要处理很多请求，我们的队列应该有 7 个容量。请记住，你增加队列的大小，它需要更长的时间才能通过系统！你实际上在延迟交易系统利用率。

利特尔法则无法预知的情况是处理请求的失败。请记住，如果由于某种原因你的管道发生混乱，你将丢失队列中的所有请求。这可能使重新创建请求变得很困难或者不会发生，这是需要防范的。为了缓解这种情况，你可以坚持队列大小为零，或者可以转移到一个持久队列，这只是一个队列，可以在需要时随时读取的地方保存。

队列在你的系统中可能很有用，但由于它的复杂性，它通常是我建议实现的最后的一个优化手段。

context 包

正如我们所看到的，在并发程序中，由于超时，取消或系统其他部分的故障往往需要抢占操作。我们已经看过了创建 done channel 的习惯用法，该 channel 在你的程序中流动并取消所有阻塞的并发操作。这很好，但它也是有限的。

如果我们可以在简单的通知上附加传递额外的信息以取消：为什么取消发生，或者我们的函数是否有需要完成的最后期限（超时），这将非常有用。

事实证明，对于任何规模的系统来说，使用这些信息来包装已完成的频道是非常常见的，因此 Go 语言的作者们决定为此创建一个标准模式。它起源于一个在标准库之外的实验功能，但是在 Go 1.7 中，context 包被引入标准库中，这使得它成为考虑并发问题时的一个标准的风格。

如果看一下上下文包，我们看到它非常简单：

```
var Canceled = errors.New("context canceled")
var DeadlineExceeded error = deadlineExceededError{}

type CancelFunc
type Context

func Background() Context
func TODO() Context
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
func WithDeadline(parent Context, deadline time.Time) (Context, CancelFunc)
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
func WithValue(parent Context, key, val interface{}) Context
```

我们稍后会重新讨论这些类型和函数，但现在让我们关注 context 类型。这是通过系统流动的类型，就像 done channel 一样。如果使用上下文包，那么位于顶级并发调用下游的每个函数都会将 context 作为其第一个参数。类型如下所示：

```
type Context interface {

// 当为该 context 工作的 work 被取消时，deadline 将返回时间。在没有设定期限的情况下，
//deadline 返回 ok==false。连续的调用 deadline 返回相同的结果
Deadline() (deadline time.Time, ok bool)

// 当为该 context 工作的 work 被取消时，返回一个关闭的 channel。如果这个 context
// 不能被取消，那么 Done 可能返回 nil。连续调用完成返回相同的值
Done() <-chan struct{}

// Err 在完成后返回一个 non-nil 值。如果 context 被取消，或者在 context
// 的 deadline 结束时，如果 context 被取消，Err 将被取消。没有定义 Err 的其他值。连续调
// 结束后，用 Err 返回相同的值
Err() error

// 值返回与此 context 关联的 key 或者 nil，如果没有与键关联的值，则返回值为 nil。
```

```
    // 连续调用具有相同 key 的值将返回相同的结果
    Value(key interface{}) interface{}
    }
```

这看起来也很简单。有一个 `Done` 方法返回当我们的函数被抢占时关闭的 channel。还有一些新的但易于理解的方法：一个 `Deadline` 函数，用于指示在一定时间之后 goroutine 是否会被取消，以及一个 `Err` 方法，如果 goroutine 被取消，将返回非零。但 `Value` 方法看起来似乎有点不合适。这是为了什么呢？

Go 语言作者们注意到，goroutine 的主要用途之一是为请求提供服务的程序。通常在这些程序中，除了抢占信息之外，还需要传递特定于请求的信息。这是 `Value` 函数的目的。我们稍后会进一步讨论这个问题，但现在我们只需要知道上下文包有两个主要目的：

- 提供一个可以取消你的调用图中分支的 API。
- 提供用于通过呼叫传输请求范围数据的数据包。

让我们关注第一个方面：取消。

正如我们在本章前面“防止 goroutine 泄漏”中所学到的，函数中的取消有三个方面：

- goroutine 的父 goroutine 可能想要取消它。
- 一个 goroutine 可能想要取消它的子 goroutine。
- goroutine 中的任何阻塞操作都必须是可抢占的，以便它可以被取消。

context 包帮助管理所有这三个东西。

正如我们所提到的，`context` 类型将是你的函数的第一个参数。如果你看看 `context` 接口上的方法，你会发现没有任何东西可以改变底层结构的状态。此外，接收 `context` 的函数并不能取消它。这保护了调用堆栈上的函数被子

函数取消上下文的情况。结合 done channel 提供的完成函数，这允许上下文类型安全地管理其前件的取消。

这就产生了一个问题：如果 context 是不可变的，那么我们如何影响调用堆栈中当前函数下面的函数中的取消行为？

这是 context 包中的功能变得重要的地方。让我们再看看其中的几个，来刷新我们的印象：

```
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
func WithDeadline(parent Context, deadline time.Time) (Context, CancelFunc)
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
```

请注意，所有这些函数都接受一个 Context 参数，并且返回一个 Context。其中一些还有其他的参数，如截止时间和超时参数。这些函数都使用与这些函数相关的选项来生成 Context 的新实例。

WithCancel 返回一个新的 Context，它在调用返回的 cancel 函数时关闭其 done channel。WithDeadline 返回一个新的 Context，当机器的时钟超过给定的最后期限时，它关闭完成的 channel。WithTimeout 返回一个新的 Context，它在给定的超时时间后关闭其完成的 channel。

如果你的函数需要以某种方式在调用图中取消它后面的函数，它将调用其中一个函数并传递给它的上下文，然后将返回的上下文传递给它的子元素。如果你的函数不需要修改取消行为，那么函数只传递给定的上下文。

通过这种方式，调用图的连续图层可以创建符合其需求的上下文，而不会影响其父母节点。这为如何管理调用图的分支提供了一个非常可组合的优雅解决方案。

context 包就是本着这种精神来串联起你程序的调用图的。在面向对象的范例中，通常将对经常使用的数据的引用存储为成员变量，但重要的是不要使

用 context.Context 的实例来执行此操作。context.Context 的实例可能与外部看起来相同，但在内部它们可能会在每个栈帧更改。出于这个原因，总是将 context 的实例传递给你的函数是很重要的。通过这种方式，函数具有用于它的上下文，而不是用于堆栈 N 的上下文。

在异步调用图的顶部，你的代码可能不会传递上下文。要启动链，上下文包提供了两个函数来创建上下文的空实例：

```
func Background() Context
func TODO() Context
```

Background() 只是返回一个空的上下文。TODO() 不是用于生产，而是返回一个空的上下文。TODO() 的预期目的是作为一个占位符，当你不知道使用哪个上下文，或者你希望你的代码被提供一个上下文，但上游代码还没有提供。

所以让我们把所有这些用于使用。我们来看一个使用完成 channel 模式的例子，并且看看我们可以从切换到使用 context 包获得什么好处。这是一个同时打印问候和告别的程序：

```
func main() {
    var wg sync.WaitGroup
    done := make(chan interface{})
    defer close(done)

    wg.Add(1)
    go func() {
        defer wg.Done()
        if err := printGreeting(done); err != nil {
            fmt.Printf("%v", err)
            return
        }
    }()

    wg.Add(1)
    go func() {
        defer wg.Done()
        if err := printFarewell(done); err != nil {
            fmt.Printf("%v", err)
```

```
            return
        }
    }()

    wg.Wait()
}

func printGreeting(done <-chan interface{}) error {
    greeting, err := genGreeting(done)
    if err != nil {
        return err
    }
    fmt.Printf("%s world!\n", greeting)
    return nil
}

func printFarewell(done <-chan interface{}) error {
    farewell, err := genFarewell(done)
    if err != nil {
        return err
    }
    fmt.Printf("%s world!\n", farewell)
    return nil
}

func genGreeting(done <-chan interface{}) (string, error) {
    switch locale, err := locale(done); {
    case err != nil:
        return "", err
    case locale == "EN/US":
        return "hello", nil
    }
    return "", fmt.Errorf("unsupported locale")
}

func genFarewell(done <-chan interface{}) (string, error) {
    switch locale, err := locale(done); {
    case err != nil:
        return "", err
    case locale == "EN/US":
        return "goodbye", nil
    }
    return "", fmt.Errorf("unsupported locale")
}

func locale(done <-chan interface{}) (string, error) {
    select {
    case <-done:
        return "", fmt.Errorf("canceled")
    case <-time.After(1*time.Minute):
    }
    return "EN/US", nil
}
```


运行此代码会产生：

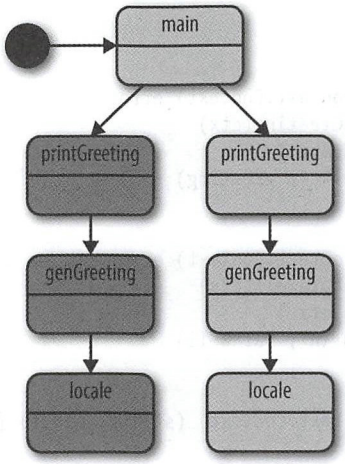
```
goodbye world!
hello world!
```

忽略竞争条件（我们可以在收到问好之前接收到我们的告别！），我们可以看到我们的程序有两个分支同时运行。我们通过创建完成通道并将其传递给我们的调用图来设置标准抢占方法。如果我们在 `main` 的任何一点关闭完成的频道，那么两个分支都将被取消。

通过引入 goroutine，我们已经开辟了以几种不同且有趣的方式来控制该程序的可能性。我们可能希望 `genGreeting` 在耗时过长的时候发生超时。也许我们不希望 `genFarewell` 调用 `locale`，如果我们知道其父函数将很快被取消。在每个堆栈框架中，一个函数可以影响其下的整个调用堆栈。

使用 `done` channel 模式，我们可以通过将传入的 `done` channel 包装到其他 `done` channel 中，然后在其中任何一个通道启动时返回，但我们不会获得关于 `Context` 给我们的最后期限和错误的额外信息。

为了比较 `done` channel 模式和使用 `context` 包谁更容易，让我们用树来表示这个程序。树中的每个节点代表一个函数的调用。



让我们修改我们的程序，使用 context 包而不是 done channel。因为我们现在具有 context.Context 的灵活性，所以我们可以引入一个有趣的场景。

假设 genGreeting 只想在放弃调用 locale 之前等待 1s，超时时间为 1s。我们也想要在 main 函数中建立一些智能逻辑。如果 printGreeting 不成功，我们也想取消我们对 printFarewall 的调用。毕竟，如果我们不打声招呼，说再见就没有意义了！

使用 context 包实现这一点很简单：

```
func main() {
    var wg sync.WaitGroup
    ctx, cancel := context.WithCancel(context.Background())❶
    defer cancel()
    wg.Add(1)
    go func() {
        defer wg.Done()
        if err := printGreeting(ctx); err != nil {
            fmt.Printf("cannot print greeting: %v\n", err)
            cancel()❷
        }
    }()
    wg.Add(1)
    go func() {
        defer wg.Done()
        if err := printFarewell(ctx); err != nil {
            fmt.Printf("cannot print farewell: %v\n", err)
        }
    }()
    wg.Wait()
}
func printGreeting(ctx context.Context) error {
    greeting, err := genGreeting(ctx)
    if err!=nil { return err }
    fmt.Printf("%s world!\n", greeting)
    return nil
}
func printFarewell(ctx context.Context) error {
    farewell, err := genFarewell(ctx)
    if err!=nil { return err }
    fmt.Printf("%s world!\n", farewell)
    return nil
}
func genGreeting(ctx context.Context) (string, error) {
```


```
    ctx, cancel := context.WithTimeout(ctx, 1*time.Second) ❸
    defer cancel()
    switch locale, err := locale(ctx); {
    case err != nil:
        return "", err
    case locale ==  "EN/US":
        return "hello", nil
    }
    return "", fmt.Errorf("unsupported locale")
}
func genFarewell(ctx context.Context) (string, error) {
    switch locale, err := locale(ctx); {
    case err != nil:
        return "", err
    case locale == "EN/US":
        return "goodbye", nil
    }
    return "", fmt.Errorf("unsupported locale")
}
func locale(ctx context.Context) (string, error) {
    select {
    case <-ctx.Done():
        return "", ctx.Err()❹
    case <-time.After(1 * time.Minute):
    }
    return "EN/US", nil
}
```

❶ 这里 main 用 context.Background() 创建一个 Context 并用 context.WithCancel 包装它以允许取消。

❷ 在这一行上，如果从打印问候语返回错误，main 将取消上下文。

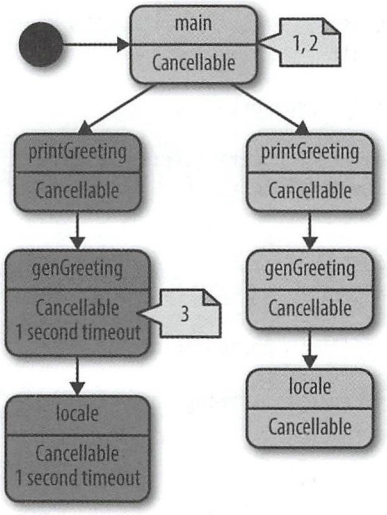
❸ 这里 genGreeting 用 context.WithTimeout 包装它的 Context。这将在 1 秒后自动取消返回的上下文，从而取消它传递上下文的任何子函数，即语言环境。

❹ 这一行返回为什么 Context 被取消的原因。这个错误会一直冒泡到 main，这会导致取消。

以下是运行此代码的结果：

```
cannot print greeting: context deadline exceeded
cannot print farewell: context canceled
```

让我们使用我们的调用图了解发生了什么。这里的数字对应于前面例子中的代码标注。



我们可以从我们的输出中看到该系统运行的很完美。由于我们确保 `local` 至少需要一分钟来运行，因此我们在 `genGreeting` 中的调用将始终超时，这意味着 `main` 会始终取消 `printFarewell` 下面的调用图。

请注意，`genGreeting` 是如何构建自定义的 `Context.Context` 以满足其需求，而不必影响父级的 `context`。如果 `genGreeting` 成功返回，并且 `printGreeting` 需要再次调用，则可以在不泄漏有关 `genGreeting` 如何操作的信息的情况下进行。这种可组合性使你能够编写大型系统，而无需在整个调用图中混淆问题。

我们可以对这个程序进行另一个改进：因为我们知道 `locale` 需要大约一分钟的时间来运行，所以我们可以检查是否给了我们最后期限，如果是的话，我们是否会遇到它。这个例子演示了如何使用 `context.Context` 的 `Deadline` 方法：

```
func main() {
    var wg sync.WaitGroup
```

```
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()

    wg.Add(1)
    go func() {
        defer wg.Done()

        if err := printGreeting(ctx); err != nil {
            fmt.Printf("cannot print greeting: %v\n", err)
            cancel()
        }
    }()

    wg.Add(1)
    go func() {
        defer wg.Done()
        if err := printFarewell(ctx); err != nil {
            fmt.Printf("cannot print farewell: %v\n", err)
        }
    }()

    wg.Wait()
}

func printGreeting(ctx context.Context) error {
    greeting, err := genGreeting(ctx)
    if err !=nil { return err }
    fmt.Printf("%s world!\n", greeting)
    return nil
}

func printFarewell(ctx context.Context) error {
    farewell, err := genFarewell(ctx)
    if err !=nil { return err }
    fmt.Printf("%s world!\n", farewell)
    return nil
}

func genGreeting(ctx context.Context) (string, error) {
    ctx, cancel := context.WithTimeout(ctx, 1*time.Second)
    defer cancel()
    switch locale, err := locale(ctx); {
    case err != nil:
        return "", err
    case locale ==  "EN/US":
        return "hello", nil
    }
    return "", fmt.Errorf("unsupported locale")
}
```


```
func genFarewell(ctx context.Context) (string, error) {
    switch locale, err := locale(ctx); {
    case err != nil:
        return "", err
    case locale == "EN/US":
        return "goodbye", nil
    }
    return "", fmt.Errorf("unsupported locale")
}

func locale(ctx context.Context) (string, error) {
    if deadline, ok := ctx.Deadline(); ok {❶
        if deadline.Sub(time.Now().Add(1*time.Minute)) <= 0 {
            return "", context.DeadlineExceeded
        }
    }
    select {
    case <-ctx.Done():
      return "", ctx.Err()
    case <-time.After(1 * time.Minute):
    }
    return "EN/US", nil
}
```

❶ 在这里我们检查我们的上下文是否提供了截止。如果确实如此，并且我们的系统时钟已超过截止时间，那么我们只会返回上下文包中定义的特定错误，即 DeadlineExceeded。

虽然这个迭代程序的差异很小，但它允许 locale 函数快速失败。在调用下一个函数的成本很高的程序中，这可能会节省大量的时间，但至少它也允许该函数立即失败，而不必等待实际的超时发生。唯一的问题是，你必须知道你的下级调用图需要多长时间，这个实践起来可能非常困难。

这将我们带到了上下文包提供的另一半功能：用于存储和检索请求范围数据的 Context 的数据包。请记住，当一个函数创建一个 goroutine 和 Context 时，它通常会启动一个将为请求提供服务的 goroutine，并且进一步向下的函数可能需要有关请求的信息。以下是如何在上下文中存储数据以及如何检索数据的示例：

```
func main() {
    ProcessRequest("jane", "abc123")
}

func ProcessRequest(userID, authToken string) {
    ctx := context.WithValue(context.Background(), "userID", userID)
    ctx = context.WithValue(ctx, "authToken", authToken)
    HandleResponse(ctx)
}

func HandleResponse(ctx context.Context) {
    fmt.Printf(
        "handling response for %v (%v)",
        ctx.Value("userID"),
        ctx.Value( “authToken"),
    )
}
```

输出如下：

```
handling response for jane (abc123)
```

很简单的东西。唯一的限制条件是：

- 你使用的键值必须满足 Go 语言的可比性概念，也就是运算符 == 和 != 在使用时需要返回正确的结果。
- 返回值必须安全，才能从多个 goroutine 访问。

由于 Context 的键和值都被定义为 interface{}，所以当试图检索值时，我们会失去 Go 语言的类型安全性。key 可以是不同的类型，或者与我们提供的 key 略有不同。值可能与我们预期的不同。出于这些原因，Go 语言作者建议你在从 Context 中存储和检索值时遵循一些规则。

首先，他们建议你在软件包中定义一个自定义键类型。只要其他软件包执行相同的操作，则可以防止上下文中的冲突。作为一个提醒，为什么让我们看看一个简短的程序，试图将键存储在具有不同类型的映射中，但具有相同的基础值：

```
type foo int
type bar int

m := make(map[interface{}]int)
m[foo(1)] = 1
m[bar(1)] = 2

fmt.Printf("%v", m)
```

输出如下：

```
map[1:1 1:2]
```

你可以看到，虽然基础值是相同的，但不同的类型信息在 map 中区分它们。由于你为软件包 key 定义的类型未导出，因此其他软件包不能与你在软件包中生成的 key 冲突。

由于我们不导出用于存储数据的 key，因此我们必须导出为我们检索数据的函数。这能很好地工作，因为它允许这些数据的使用者使用静态的、类型安全的函数。

当你把所有这些放在一起时，你会得到类似下面的这个例子：

```
func main() {
    ProcessRequest("jane", "abc123")
}

type ctxKey int

const (
    ctxUserID ctxKey = iota
    ctxAuthToken
)

func UserID(c context.Context) string {
    return c.Value(ctxUserID).(string)
}

func AuthToken(c context.Context) string {
    return c.Value(ctxAuthToken).(string)
}
```


```
func ProcessRequest(userID, authToken string) {
    ctx := context.WithValue(context.Background(), ctxUserID, userID)
    ctx = context.WithValue(ctx, ctxAuthToken, authToken)
    HandleResponse(ctx)
}

func HandleResponse(ctx context.Context) {
    fmt.Printf(
        "handling response for %v (auth: %v)",
        UserID(ctx),
        AuthToken(ctx),
    )
}
```

运行此代码会产生：

```
handling response for jane (auth: abc123)
```

我们现在有一种类型安全的函数来从 context 获取值，如果消费者在不同的包中，他们不会知道或关心用于存储信息的 key。但是，这种技术确实会造成问题。

在前面的例子中，我们假设 HandleResponse 确实存在于另一个名为 response 的包中，我们假设 ProcessRequest 包位于名为 pross 的包中。process 包必须导入响应包才能调用 HandleResponse，但 HandleResponse 无法访问 process 包中定义的访问器函数，因为导入过程会形成循环依赖关系。由于用于在 Context 中存储 key 的类型对于 process 包来说是私有的，所以 responce 包无法检索这些数据！

这迫使体系结构创建以从多个位置导入的数据类型为中心的包。这当然不是一件坏事，但有些事情得注意。

context 包非常整洁，但尚未被整个 Go 语言社区所赞美。在 Go 语言社区中，context 包一直存在争议。建议取消 context 包的提议已经相当受欢迎，而且在 context 中存储任意数据的能力以及存储数据的类型不安全的方式造成了一些分歧。虽然已经减少了部分访问函数缺乏类型安全性，但是仍然可以

通过存储不正确的类型来引入错误。然而，更大的问题肯定是开发人员应该在 context 的实例中存储什么的问题。

关于什么是适当的、最普遍的指导，下面是上下文包中的下面有点含糊的注释：

> 仅将上下文值用于传输进程和请求的请求范围数据，API 边界，而不是将可选参数传递给函数。

很清楚什么是可选参数（你不应该使用 context 来满足你对 Go 语言支持可选参数的秘密愿望），但什么是“请求范围数据”？据说它“转换进程和API边界”，但它可以描述很多东西。我发现定义它的最好方法是与团队一起提出一些启发式方法，并在代码评审中评估它们。这是我的启发式：

1. *数据应该通过进程或 API 边界。*
   如果你在进程的内存中生成数据，那么除非你通过 API 边界传递数据，否则可能不是一个很好的选择。
2. *数据应该是不可变的。*
   如果不是，那么根据定义，你存储的内容不是来自请求的内容。
3. *数据应趋向简单类型。*
   如果请求范围数据是为了传递进进程和 API 边界，那么如果另一方不需要导入一个复杂的包的图，那么将这些数据拉出就容易得多。
4. *数据应该是数据，而不是类型与方法。*
   操作是逻辑的，属于消耗这些数据的东西。
5. *数据应该有助于修饰操作，而不是驱动它们。*
   如果你的算法根据 context 中包含或不包含的内容而有所不同，你可能会跨越可选参数的范围。

这些不是硬性规定，它们是启发式的。但是，如果你发现存储在上下文中的数据违反了上述所有五条准则，你可能需要仔细观察下你正在选择去做的是什么。

需要考虑的另一个方面是该数据在使用之前可能需要经过多少层。如果在接收数据的地方和使用地点之间有几个框架和几十个函数，你是否想要倾向于冗长的自解释的参数名，并将数据作为参数添加？或者你更愿意将它放在`context`中，从而创建一个不可见的依赖关系？每种方法都有优点，最终这是你和你的团队必须做出的决定。

即使采用这些启发式方法，值是否是请求范围数据仍然是一个难以回答的问题。看看下面的表格。它列出了我对每种数据是否满足我列出的五种启发式的看法。你同意吗？

数据	1	2	3	4	5
Request ID	√	√	√	√	√
UserID	√	√	√	√	
URL	√	√			
API Server Connection					
Authorization Token	√	√	√	√	
Request Token	√	√	√		

有时很明显，某些内容不应该存储在上下文中，因为它与API服务器连接有关，但有时不太清楚。什么是授权令牌？它是不可变的，它可能是一部分字节，但是这些数据的接收者不会使用它来确定是否要求字段？这些数据是否属于上下文？为了进一步浑浊水域，一个团队可以接受的是另一个团队可能不会接受的。

总而言之，这个问题并没有一个一概而就的答案。该软件包已被纳入标准库，因此你必须对其使用形成一些意见，但该意见可能（也可能应该）根据你所触及的项目而改变。我留给你的最后建议是context提供的取消功能非常有用，你对数据包的感受不应该阻止你使用它。

小结

本章介绍了很多内容。我们结合了 Go 语言的并发原语来形成模式，以帮助编写可维护的并发代码。既然你熟悉了这些模式，我们可以讨论如何将这些模式合并到其他模式中，以帮助你编写大型系统。下一章将会给你一个关于这样做的技术的概述。

第 5 章

大规模并发

之前的章节中，你已经学习了一些 Go 语言中使用并发的常见模式。现在让我们将重点转移到这些模式的使用上，我们会利用这些模式构建一系列的并发系统，这些实践可以帮助你写出规模化、组件化的系统。

在这一章，我们将研究如何使用单进程进行大规模并发操作，并开始探寻多进程处理时并发是如何发挥作用的。

异常传递

编写并发代码，特别是在分布式系统中，你的系统中非常容易出现一些奇怪问题，并且难以理解为什么会发生这种情况。为了将你自己、你的团队、你的用户从众多的痛苦中拯救出来，你需要仔细考虑异常（error）是如何通过分布式系统传递的，以及问题最终将如何呈现给使用者。在第 4 章“异常处理”中，我们研究了如何将 goroutine 中的异常传递出来，但是并未过多探讨这些异常应当是什么样的，或者异常应当如何经过一个庞大而复杂的系统传递出来。以下是用于处理并发系统中异常的一个示意框架，让我们用这个框架来对异常传递的哲学进行一些研究。

许多开发人员有个误解，认为在系统流程中异常的传递并不是那么重要。他们通常会谨慎的考虑数据会如何经过系统，但是却轻易的容忍异常，未经过思考就将异常从栈中抛出，最终导致异常直接展示在了用户面前。Go 语言试

图纠正这种不良习惯，强制开发人员处理调用栈上的每个关键点的异常，但是在系统控制流中将异常视为不太重要仍然是一种常见的行为。其实只需要一点计划和非常小的代价，你就可以将异常控制在系统范围内，优化你的用户体验。

首先让我们来明确异常是什么，什么时候发生，提供了那些好处。

出现异常表示着你的系统进入了一个无法满足用户操作的状态，这个操作可能是显式的，也可能是隐式的。这时系统需要传达几个关键的信息：

发生了什么

这部分异常信息包含了对异常事件的描述。例如：“磁盘已满”，“连接被重置”，“证书过期”。这些信息可能是被一些代码隐式的表达出来的，你可以用一些上下文来修饰这些信息来帮助用户理解发生了什么问题。

发生在什么时间、什么位置

异常应当总是包含完整的栈轨迹信息，从调用的启动方式开始，以异常的实例结尾。栈轨迹信息不应该包含在异常消息中（这一点尤为重要），但当需要处理栈中的异常时应该很容易被找到。

更进一步讲，异常应当包含有关其内部运行的上下文信息。例如，在分布式系统中，异常应该有一些字段用来识别发生异常的机器。发生异常后，这些信息会对你诊断系统故障原因非常有价值。

此外，异常还应包含对应机器上的时间，并且最好是 UTC 时间。

对用户友好的信息

应当对展现给用户的异常信息进行自定义，以适应你的系统和用户。这些信息应该只包含前两点的概述以及相关信息。对用户友好的信息是从用户的角度考虑，给出一些信息，说明这些问题是否是暂时的，并且最好是一行以内的文本。

告诉用户如何获得更多的信息

在某些情况下，用户希望知道当异常发生时，具体发生了哪些故障。展现给用户的异常信息应当提供一个 ID，利用这个 ID 可以查询到对应的详细

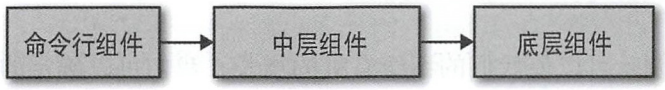
日志。这个详细日志应显示异常的完整信息：发生异常的时间（而不是异常记录的时间），异常创建时完整的堆栈调用。包含一个堆栈轨迹的 hash 也有助于聚合这些异常，就像 bug 追踪器那样跟踪问题。

默认状态下，如果你不介入，异常信息不会包含上述所有的信息。因此，你应当保持这样一种观念，任何展现给用户的异常信息如果没包含这些信息，不是出错了就是有 bug。这引出了一个可以用来处理异常的通用模型。所有的异常都几乎都能归为以下两种分类之一：

- Bug。
- 已知信息（例如：网络连接断开，磁盘写入失败等）。

Bug 是一些你未在你的系统中定义的异常，或者一些“原生”的异常，就是那些极少遇到的情况。有时这是有意为之的，在你系统最初的几次迭代中，一些罕见问题展现给用户是可以接受的。还有些时候这是意外发生的。总之，如果你同意我所提出的方法，即“原生”异常总是 bug。在确定如何传播异常时，在系统随着时间的推移如何增长以及最终向用户展示什么时，这种区别被证明是非常有用的。

想象一个多模块的大型系统：



我们假设在“底层组件”中产生了一个结构良好的异常，它正在等待上传到栈中。在“底层组件” 的背景中，这个异常的结构可能是良好的，但是在我们的系统中，它可能并非如此。让我们以每个组件的边界为例，所有传入的异常信息都必须使用我们的代码重新格式化。例如，我们在“中层组件”调用“底层组件”中可能会出错函数时，我们可以这样写：

```
func PostReport(id string) error {
    result, err := lowlevel.DoWork()
    if err != nil {
        if _, ok := err.(lowlevel.Error); ok { ❶
            err = WrapErr(err, "cannot post report with id %q", id) ❷
        }
        return err
    }
    // ...
}
```

❶ 在这里，我们检查一下接收到的异常信息，以确保它的结构是良好的。如果不是，我们就简单地将异常推到栈上，以显示出这个 bug。

❷ 在这里，我们使用一个假设的函数将传入的异常和模块相关的信息封装起来，并赋予它一个新的类型。请注意，包装异常可能会隐藏一些底层的细节，这些细节对于用户来说可能并不重要。

当异常被最初实例化时，异常信息中包含了太多的底层细节，这些细节与异常产生的根源（例如，goroutine，机器，栈轨迹等）息息相关。不过我们的体系规定了，我们应当在模块的边界处将这些底层异常转换成我们这一层模块的异常结构，并将底层细节改写成与我们模块相关的信息。现在，如果出现了一些异常，并且并非我们模块的异常结构，便可以认为是格式异常，或者是一个 bug。请注意，只在必要时才用这种方式包装异常，比如你自己的模块边界（公共函数 / 方法）或者你的代码要添加有意义的上下文。这可以防止在大量的代码中重新包装异常信息。

采用这一机制，可以使我们的系统有机的增长。我们可以确定的是，传入的异常是完整的。反过来说，我们可以保证我们正在考虑异常如何传出我们的模块。异常的正确性成为了我们系统的一个关键属性。我们也从一开始就尽可能完美、明确地处理不规范的异常，这样我们构建出一个处理异常的框架，随着时间的推移不断的纠正异常。再通过呈现给用户的类型划分，可以更清晰的划分异常。

正如我们上面说的，所有的异常都应该尽可能的记录下来。但是，当向用户显示异常时， bug 和已知罕见问题还是有一定区别的。

当我们面向用户部分的代码收到一个格式良好的异常信息时，我们知道在代码的各个层面上，我们都小心的处理了异常，我们可以将其记录下来并打印出来供用户查看。确保异常类型的准确有效是非常重要的。

当不规范的异常或 bug 传递给用户时，我们也应该记录异常，但是应该向用户显示一条友好的消息，指出发生了意外的事情。如果我们在系统中支持自动的异常报告，则应该将这些问题报告为 bug 。如果我们不这样做，我们应该建议用户提交一个 bug 反馈。请注意，不规范的异常实际上也可能包含有用的信息，但我们不能保证这一点，我们唯一能确认的是异常没有经过我们格式化。因此我们应该直截了当地展示一段人类可解读的信息，来展示刚刚发生的事情。

请记住，在这两种情况下，如果出现格式不规范的异常，我们将在消息中包含一个日志 ID，以便用户在需要更多信息时可以查询到相关的内容。因此，如果 bug 确实包含了有用的信息，有需要的用户仍然有可追踪的线索。

我们来看一个完整的例子。这个例子并不是非常健壮（例如，异常类型可能很简单），调用顺序也是线性的，这使我们只需要在模块边界处包装异常。而且，在一本书中很难用不同的包来表示函数，所以我们会假定这些函数是存在的。

首先，我们来创建一个异常类型，它包含了一个格式良好的异常应有的内容：

```
type MyError struct {
    Inner           error
    Message         string
    StackTrace      string
    Misc                    map[string]interface{}
}

func wrapError(err error, messagef string, msgArgs ...interface{}) MyError {
    return MyError{
        Inner:              err, ❶
        Message:            fmt.Sprintf(messagef, msgArgs...),
        StackTrace:   string(debug.Stack()), ❷
        Misc:               make(map[string]interface{}), ❸
```

```
    }
}

func (err MyError) Error() string {
    return err.Message
}
```

❶ 在这里，我们存储了我们正在包装的异常。通常我们会希望能够找到最底层的异常，以便在需要时可以调查发生的异常。

❷ 这行代码在创建异常时记录堆栈轨迹。过于复杂的错误类型经过 `wrapError` 包装后可能会省略一些栈帧。

❸ 在这里，我们创建一个可以存储各种杂项的变量。我们可以将并发 ID，堆栈轨迹的 hash 或可能有助于诊断异常的其他上下文信息存储在这里。

接下来，我们来创建一个底层模块：

```
// lowLevel 模块

type LowLevelErr struct {
    error
}

func isGloballyExec(path string) (bool, error) {
    info, err := os.Stat(path)
    if err != nil {
        return false, LowLevelErr{(wrapError(err, err.Error()))} ❶
    }
    return info.Mode().Perm()&0100 == 0100, nil
}
```

❶ 在这里，我们用自定义的异常来调用 `os.Stat` 中的原始异常。在这种情况下，我们可以用这个异常传递信息，而不用对它做任何修饰。

然后，让我们创建一个中间模块，用来调用底层模块的 package：

```
// intermediate 模块

type IntermediateErr struct {
    error
}
```

```
func runJob(id string) error {
    const jobBinPath = "/bad/job/binary"
    isExecutable, err := isGloballyExec(jobBinPath)

    if err != nil{
        return err ❶
    } else if isExecutable == false {
        return wrapError(nil, "job binary is not executable")
    }

    return exec.Command(jobBinPath, "--id="+id).Run() ❶
}
```

❶ 这里我们传递来自底层模块的异常。因为我们的体系结构决定，我们需要考虑从其他模块传递来的错误，而不是将它们用我们自己的错误类型包装，这里会存在一些问题，后面会提到。

最后，让我们创建一个外层的 `main` 函数来调用中间层。这是我们程序面向用户的部分：

```
func handleError(key int, err error, message string) {
    log.SetPrefix(fmt.Sprintf("[logID: %v]: ", key))
    log.Printf("%#v", err) ❸
    fmt.Printf("[%v] %v", key, message)
}

func main() {
    log.SetOutput(os.Stdout)
    log.SetFlags(log.Ltime|log.LUTC)
    err := runJob("1")
    if err != nil {
        msg := "There was an unexpected issue; please report this as a bug."
        if _, ok := err.(IntermediateErr); ok {❶
                msg = err.Error()
        }
        handleError(1, err, msg) ❷
    }
}
```

❶ 在这里，我们检查一下异常是否是预期的类型。如果是，那么可以确定这是一个结构完整的异常，我们只要简单地将其中的消息传递给用户即可。

❷ 在这一行，我们将日志和异常消息与一个 ID 绑定在一起。我们可以使用一个自增 ID，或者用 GUID 来保证 ID 的唯一性。

❸ 在这里，我们记录下异常的所有内容，以备有人需要深入了解发生的事情。

运行上面的程序，我们会得到一条日志信息，内容如下：

```
[logID: 1]: 21:46:07 main.LowLevelErr{error:main.MyError{Inner:
(*os.PathError)(0xc4200123f0),
Message:"stat /bad/job/binary: no such file or directory",
StackTrace:"goroutine 1 [running]:
runtime/debug.Stack(0xc420012420, 0x2f, 0xc420045d80)
    /home/kate/.guix-profile/src/runtime/debug/stack.go:24 +0x79
main.wrapError(0x530200, 0xc4200123f0, 0xc420012420, 0x2f, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, ...)
    /tmp/babel-79540aE/go-src-7954NTK.go:22 +0x62
main.isGloballyExec(0x4d1313, 0xf, 0xc420045eb8, 0x487649, 0xc420056050)
    /tmp/babel-79540aE/go-src-7954NTK.go:37 +0xaa
main.runJob(0x4cfada, 0x1, 0x4d4c35, 0x22)
    /tmp/babel-79540aE/go-src-7954NTK.go:47 +0x48
main.main()
    /tmp/babel-79540aE/go-src-7954NTK.go:67 +0x63
", Misc:map[string]interface {}{}}}
```

还有一条包含在标准输出里的消息：

```
[1] There was an unexpected issue; please report this as a bug.
```

在日志中我们可以看出，上面某些地方在遇到这些错误路径时处理不当，因为我们不能确定这些异常信息是否适合用户阅读，所以我们在标准输出中打印出一个简单的异常，指出发生了意外情况。现在我们回顾一下中间层，我们回想一下这个问题：我们为什么没有包装来自 lowlevel 模块的异常。我们来修正一下，看看会发生什么：

```
// intermediate 模块

type IntermediateErr struct {
    error
}

func runJob(id string) error {
    const jobBinPath = "/bad/job/binary"
    isExecutable, err := isGloballyExec(jobBinPath)
    if err != nil {
        return IntermediateErr{wrapError(
            err,
```

```
                "cannot run job %q: requisite binaries not available",
                id,
            )} ❶
        } else if isExecutable == false {
            return wrapError(
                nil,
                "cannot run job %q: requisite binaries are not executable",
                id,
            )
        }

        return exec.Command(jobBinPath, "--id="+id).Run()
    }
```

❶ 在这里，我们使用精心设计的异常信息。在这种情况下，我们想隐藏异常的底层细节，因为我们觉得这对我们模块的调用者来说并不重要。

```
func handleError(key int, err error, message string) {
    log.SetPrefix(fmt.Sprintf("[logID: %v]: ", key))
    log.Printf("%#v", err)
    fmt.Printf("[%v] %v", key, message)
}

func main() {
    log.SetOutput(os.Stdout)
    log.SetFlags(log.Ltime|log.LUTC)

    err := runJob("1")
    if err != nil {
        msg := "There was an unexpected issue; please report this as a bug."
        if _, ok := err.(IntermediateErr); ok {
            msg = err.Error()
        }
        handleError(1, err, msg)
    }
}
```

现在我们运行更新后的代码，我们得到了下面这样的日志消息：

```
[logID: 1]: 22:11:04 main.IntermediateErr{error:main.MyError
{Inner:main.LowLevelErr{error:main.MyError{Inner:(*os.PathError)
(0xc4200123f0), Message:"stat /bad/job/binary: no such file or directory",
StackTrace:"goroutine 1 [running]:
runtime/debug.Stack(0xc420012420, 0x2f, 0x0)
    /home/kate/.guix-profile/src/runtime/debug/stack.go:24 +0x79
main.wrapError(0x530200, 0xc4200123f0, 0xc420012420, 0x2f, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, ...)
```

```
    /tmp/babel-79540aE/go-src-7954DTN.go:22 +0xbb
main.isGloballyExec(0x4d1313, 0xf, 0x4daecc, 0x30, 0x4c5800)
    /tmp/babel-79540aE/go-src-7954DTN.go:39 +0xc5
main.runJob(0x4cfada, 0x1, 0x4d4c19, 0x22)
    /tmp/babel-79540aE/go-src-7954DTN.go:51 +0x4b
     main.main()
    /tmp/babel-79540aE/go-src-7954DTN.go:71 +0x63
", Misc:map[string]interface {}{}}}, Message:"cannot run job \"1\":
requisite binaries not available", StackTrace:"goroutine 1 [running]:
runtime/debug.Stack(0x4d63f0, 0x33, 0xc420045e40)
    /home/kate/.guix-profile/src/runtime/debug/stack.go:24 +0x79
main.wrapError(0x530380, 0xc42000a370, 0x4d63f0, 0x33,
0xc420045e40, 0x1, 0x1, 0x0, 0x0, 0x0, ...)
    /tmp/babel-79540aE/go-src-7954DTN.go:22 +0xbb
main.runJob(0x4cfada, 0x1, 0x4d4c19, 0x22)
    /tmp/babel-79540aE/go-src-7954DTN.go:53 +0x356
main.main()
    /tmp/babel-79540aE/go-src-7954DTN.go:71 +0x63
", Misc:map[string]interface {}{}}}
```

现在我们的异常信息正是我们希望用户看到的：

```
[1] cannot run job "1": requisite binaries not available
```

有与我们使用的方法兼容的软件包[注1]，不过你也可以使用任何异常软件包来实现这种异常处理方式。你可以设计出最高级别的异常处理，并且划分异常和精心设计的异常，然后逐步确认你创建的所有异常都是被精心设计的。

超时和取消

在并发代码运行时，超时（Timeouts）和取消（Cancellation）会频繁出现。在本节中我们将会看到，超时的处理对于创建一个易于理解的系统是至关重要的，进程被取消是其发生超时的自然反映。我们还将探讨一个并发进程可能被取消的原因。

那么，我们为什么希望我们的并发程序支持超时呢？ 这里有几个原因：

系统饱和

正如我们在第 4 章“排队”中所讨论的，如果我们的系统已经饱和（即它

注 1： 我推荐阅读 *http://github.com/pkg/errors*。

的处理请求的能力刚好足够处理），我们可能希望超出的请求返回超时，而不是花很长的时间等待响应。你采取的应对方式取决于你的问题空间，下面是一些关于何时应当超时的一般性指导：

- 请求在超时时不太可能重复。
- 没有资源来存储请求（例如，内存队列的内存，持久队列的磁盘空间）。
- 如果对系统的响应或请求发送数据有时效性的要求（我们将在接下来讨论）。如果一个请求可能会重复，超时会额外增加一个请求和超时的消耗。如果开销超过我们系统的容量，这可能会导致系统宕机。不管怎样，如果我们缺少将请求存储在队列中所需的系统资源，那也是没有意义的。即便我们符合这两个指导方针，只要我们能及时处理，让请求进入排队中意义也不大。这给我们带来了下一个支持超时的理由。

陈旧的数据

数据通常有一个窗口期，一般是在这个窗口中必须先处理更多的相关数据，或者处理数据的需求已经过期。如果一个并发进程处理数据需要的时间比这个窗口期更长，我们会想返回超时并取消并发进程。例如，如果我们的并发进程在长时间的等待之后响应请求，则在排队中的请求或其数据可能已经过时。

如果事先知道这个窗口时间，那么将 `context.WithDeadline` 或 `context.WithTimeout` 创建的 `context.Context` 传递给我们的并发进程是有意义的。如果事先不知道窗口，我们希望并发进程的父节点能够在请求不再需要时取消并发进程。`context.WithCancel` 是达到这个目的的最佳选择。

试图防止死锁

在大型系统中，尤其是分布式系统中，有时难以理解数据流动的方式，或者可能出现的罕见情况。为了保证系统不会发生死锁，建议在所有并发操作中增加超时处理。超时时间不一定要接近执行并发操作所需的实际时间。

不过超时的目的只是为了防止死锁，所以需要它足够短，使死锁的系统在合理的时间内解除阻塞即可。

以上内容使我们知道了，尝试通过设置超时可以将一个死锁系统转变为一个活锁系统。不过，在大型系统中，由于存在更多灵活的组件，在系统死锁后，你的系统更可能会遇到时序配置不同步的情况。因此，最好是在允许的时间内尽可能修复活锁，好过发生死锁后只有通过重新启动才能恢复系统。

请注意，这不是如何正确构建系统的建议，而是关于如何建立一个对时间问题有容错能力的系统，这些错误在开发和测试过程中可能不容易遇到。我建议你将超时设置在适当的位置，但是目标应该是在没有触发超时的情况下处理完所有的请求。

现在我们已经掌握应当何时使用超时了，让我们把注意力转向取消，以及如何建立一个并发处理来优雅地处理取消。并发进程可能被取消的原因有很多：

超时

超时是隐式取消。

用户干预

为了获得良好的用户体验，通常建议维持一个长链接，然后以轮询间隔将状态报告给用户，或允许用户查看他们认为合适的状态。当用户使用并发程序时，有时需要允许用户取消他们已经开始的操作。

父进程取消

对于这个问题，如果任何一种并发操作的父进程停止，那子进程也将被取消。

复制请求

我们可能希望将数据发送到多个并发进程，以尝试从其中一个进程获得更快的响应。当第一个回来的时候，我们就会取消其余的进程。我们将在第5章“复制请求”中详细讨论。

也可能有其他的原因。然而，“为何”这个问题并不像“如何”这样的问

题那么困难或有趣。在第 4 章中，我们探讨了两种取消并发进程的方法：channel done 和 context.Context 类型。这是相对容易的一部分， 在这里我们想要探索更复杂的问题：当一个并发进程被取消时，对于正在执行的算法及其下游消费者意味着什么？ 在编写可随时终止的并发代码时，需要考虑哪些事项？

为了回答这些问题，我们需要探索的第一件事是并发进程的可抢占性。观察下面的代码，并假设它在自己的 goroutine 中运行：

```
var value interface{}

select {
case <-done:
    return
case value = <-valueStream:
}

result := reallyLongCalculation(value)

select {
case <-done:
    return
case resultStream<-result:
}
```

我们已经可以从 valueStream 的读取数据然后写入 resultStream，并监听 channel done，检查 goroutine 是否被取消了，但是仍然有一些问题。reallyLongCalculation 似乎不能抢占，而且根据名字来看，可能需要很长时间！这意味着，如果在长时间计算正在执行的时候，如果有东西试图取消这个 goroutine，那么在我们确认取消和停止之前可能需要很长时间。让我们试着让 reallyLongCaluclation 支持抢占，看看会发生什么：

```
reallyLongCalculation := func(
    done <-chan interface{},
    value interface{},
) interface{} {
    intermediateResult := longCalculation(value)
    select {
    case <-done:
        return nil
    default:
    }
```

```
    return longCaluclation(intermediateResult)
}
```

我们已经取得了一些进展：reallyLongCaluclation 现在可以被抢占了，不过我们只将问题减少了一半：我们只能在 reallyLongCalculation 调用其他函数实现抢占。为了解决这个问题，我们需要编写 longCalculation，就像下面这样：

```
reallyLongCalculation := func(
    done <-chan interface{},
    value interface{},
) interface{} {
    intermediateResult := longCalculation(done, value)
    return longCaluclation(done, intermediateResult)
}
```

如果认为这个推理的结论是合乎逻辑的，那就能得出以下两个必要的任务：定义我们的并发进程可抢占的周期，确保运行周期比抢占周期长的功能本身都是可抢占的。一个简单的方法是将你的 goroutine 代码段分解成小段。你应该注意那些不可抢占的原子操作，确保它们的运行时间小于你认为可以接受的时间。

这里还有另外一个潜在的问题：如果我们的 goroutine 恰好修改了共享状态（例如数据库，文件，内存数据结构），那当 goroutine 被取消时会发生什么？你的 goroutine 会试图将这个中间状态回滚吗？回滚过程需要多长时间？goroutine 已经接收到了停止的信号，所以它不应该花太长的时间来回滚它之前的工作，对吧？

就如何处理这个问题很难给出通用的建议，因为你的算法的性质很大程度上决定了你应当如何解决这个问题。然而，如果你将对共享状态的修改保持在一个很小的范围内，并且确保这些修改很容易回滚，那么你可以很好的处理取消。如果可能的话，将中间结果存储在内存，然后尽可能快的修改状态。下面是一个错误的示范：

```
result := add(1, 2, 3)
```

```
writeTallyToState(result)
result = add(result, 4, 5, 6)
writeTallyToState(result)
result = add(result, 7, 8, 9)
writeTallyToState(result)
```

这里我们改变了三次状态。如果运行这个代码的 goroutine 在最后写入之前被取消，我们需要以某种方式回滚前两个调用来修改 `TallyToState`。对比这个方法：

```
result := add(1,2,3,4,5,6,7,8,9)
writeTallyToState(result)
```

这次我们必须回滚范围要小得多。如果在调用 `writeToState` 之后取消，我们仍然需要一种方法来撤回我们的修改，但是由于我们只修改了一次状态，发生这种情况的可能性要小得多。

你需要关心的另一个问题是重复的消息。假设你有一个管道，它有三个阶段：生成阶段，阶段 A 和阶段 B。生成阶段通过记录上一次 channel 被读取的时间，来监控阶段 A 持续的时间。如果当前实例变得不正常，则产生新的实例 A2。如果发生这种情况，阶段 B 可能会收到重复的消息（见图 5-1）。

你可以从图 5-1 看到，如果在阶段 A 已经将阶段 B 的结果发送到阶段 B 之后，接收取消的信号，那么阶段 B 可能会收到重复的消息。

有很多种方法可以避免发送重复的消息。最简单的方法（也是我推荐的方法）是让一个父 goroutine 在子 goroutine 已经发送完结果之后发送一个取消信号。这需要各阶段之间的双向通信，我们将在本章后面“心跳”中详细介绍。其他方法是：

接收到的第一个或最后一个消息

如果你的算法允许，或者你的并发进程是幂等的，那么你可以简单地在下游进程中允许可能存在的重复消息，并从接收到的第一个消息或最后一个消息中挑选一个处理。

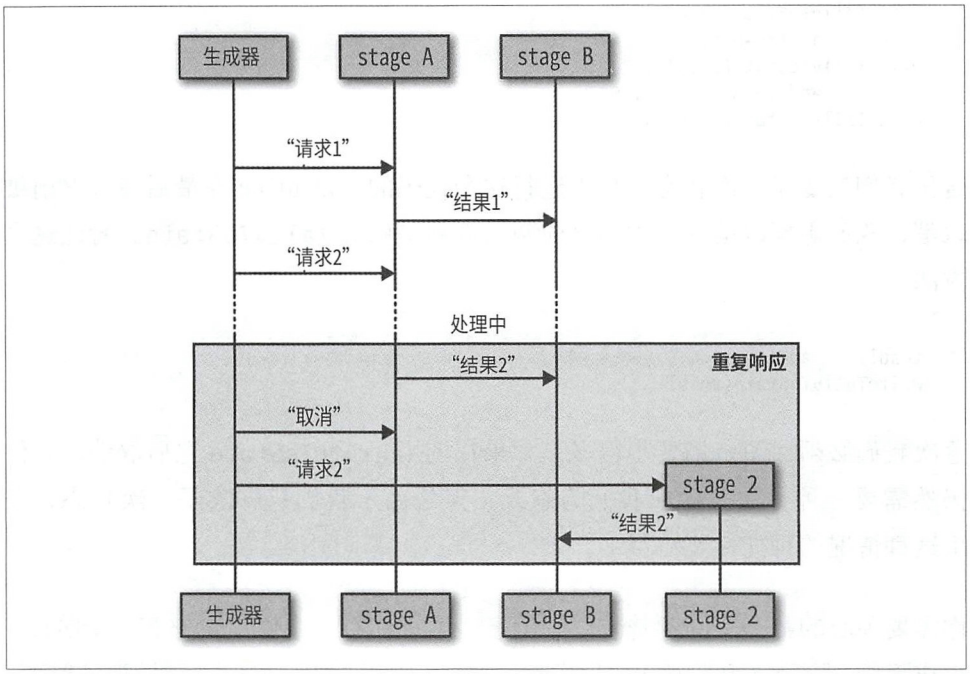


图 5-1：如何发送重复消息的示例

向父 goroutine 确认权限

你可以与你的父 goroutine 使用双向通信来确认你发送消息的权限。这种方法与心跳类似，如图 5-2 所示。

因为我们明确请求允许在 B 的 channel 上执行写入操作，这比心跳更安全；然而，在实践中很少这样做，因为它比心跳更加复杂，而心跳更普遍且有效，所以我建议你只使用心跳。

心跳

心跳是并发进程向外界发出信号的一种方式。这个说法来自人体解剖学，在解剖学中心跳反应了观察者的生命体征。心跳在 Go 语言 之前就已经存在，而且一直非常有效。

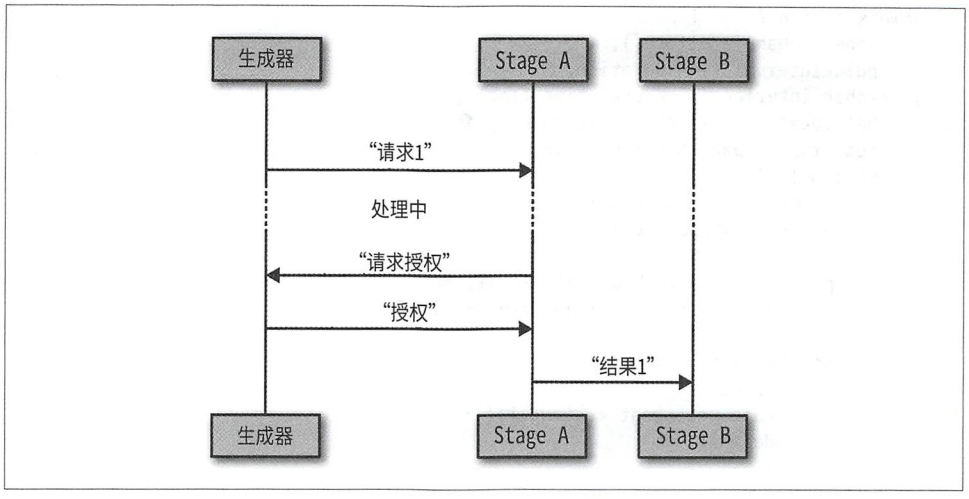


图 5-2：轮询父 goroutine 的示例

在设计并发程序时，一定要考虑到超时和取消。如果从一开始就忽略超时和取消，然后在后期尝试加入它们，这有点像在蛋糕烤好后再加鸡蛋。

在并发编程中，有几个的原因使心跳变得格外有趣。它允许我们对系统有深入的了解，当系统工作不正常时，它可以对系统进行测试。

本节将讨论两种不同类型的心跳：

- 在一段时间间隔内发出的心跳。
- 在工作单元开始时发出的心跳。

在一段时间间隔上发出的心跳对并发代码很有用，尤其是当它在处于等待状态。因为你不知道新的事件什么时候会被触发，你的 goroutine 可能会在等待某件事情发生的时候挂起。心跳是告诉监听程序一切安好的一种方式，而静默状态也是预料之中的。

下面的代码演示了一个会发出心跳的 goroutine：

```
doWork := func(
    done <-chan interface{},
    pulseInterval time.Duration,
) (<-chan interface{}, <-chan time.Time) {
    heartbeat := make(chan interface{}) ❶
    results := make(chan time.Time)
    go func() {
        defer close(heartbeat)
        defer close(results)

        pulse := time.Tick(pulseInterval) ❷
        workGen := time.Tick(2*pulseInterval) ❸

        sendPulse := func() {
            select {
            case heartbeat <-struct{}{}:
            default: ❹
            }
        }
        sendResult := func(r time.Time) {
            for {
                select {
                case <-done:
                    return
                case <-pulse: ❺
                    sendPulse()
                case results <- r:
                    return
                }
            }
        }

        for {
            select {
            case <-done:
                return
            case <-pulse: ❺
                sendPulse()
            case r := <-workGen:
                sendResult(r)
            }
        }
    }()
    return heartbeat, results
}
```

❶ 我们建立了一个发送心跳的 channel 。我们把这个返回给 `doWork`。

❷ 我们设定心跳的间隔时间为我们接到的 `pulseInterval`。每隔一个 `pulseInterval` 的时长都会有一些东西读取这个 channel。

❸ 这是另一个用来模拟滴答声的channel。我们选择的持续时间大于pulseInterval，这样我们就能看到从goroutine中发出的一些心跳。

❹ 注意，这里我们加入了一个默认语句。我们必须时刻警惕这样一个事实：可能会没有人接收我们的心跳。从goroutine发出的信息是重要的，但心跳却不一定重要。

❺ 就像done channel一样，当你执行发送或接收时，你也需要包含一个发送心跳的分支。

请注意，因为我们可能在等待输入时发出多个心跳，或者在等待发送结果时发出多个心跳，所以所有的select语句都需要在for循环中。目前为止看起来都很好；我们如何利用这个函数并消费它所发出的事件呢？让我们来看看：

```
done := make(chan interface{})
time.AfterFunc(10*time.Second, func() { close(done) }) ❶

const timeout = 2*time.Second ❷
heartbeat, results := doWork(done, timeout/2) ❸
for {
    select {
    case _, ok := <-heartbeat: ❹
        if ok == false {
            return
        }
        fmt.Println("pulse")
    case r, ok := <-results: ❺
        if ok == false{
            return
        }
        fmt.Printf("results %v\n", r.Second())
    case <-time.After(timeout): ❻
        return
    }
}
```

❶ 我们声明了一个标准的done channel，并在10秒后关闭。这给了我们的goroutine做一些工作的时间。

❷ 这里我们设置了超时时间。我们使用此方法将心跳间隔与超时时间联系起来。

❸ 我们在这里 `timeout/2`。这使我们的心跳有额外的响应时间，以便我们的超时不太敏感。

❹ 在这里，我们处理心跳。当没有消息时，我们至少知道每过 `timeout/2` 的时间会从心跳 channel 发出一条消息。如果我们什么都没有收到，我们便知道是 goroutine 本身出了问题。

❺ 在这里，我们处理 `results` channel；这里没什么特别的。

❻ 如果我们没有收到心跳或其他消息，就会超时。

运行此代码得到下面的结果：

```
pulse
pulse
results 52
pulse
pulse
results 54
pulse
pulse
results 56
pulse
pulse
results 58
pulse
```

你可以看到，我们收到的每个消息之间大约有两个心跳。

在一个功能正常的系统中，心跳并没什么有趣的。我们可能会用它们来收集关于空闲时间的统计数据，但是当你的 goroutine 不像预期的那样运行时，基于间隔的心跳的作用就会非常大。

思考下面的例子。我们将在两次迭代后停止 goroutine，但却不关闭我们的任何一个 channel，来模拟一个产生了异常的 goroutine。让我们看下代码：

```
doWork := func(
    done <-chan interface{},
    pulseInterval time.Duration,
) (<-chan interface{}, <-chan time.Time) {
    heartbeat := make(chan interface{})
```

```
    results := make(chan time.Time)
    go func() {
        pulse := time.Tick(pulseInterval)
        workGen := time.Tick(2*pulseInterval)
        sendPulse := func() {
             select {
             case heartbeat <-struct{}{}:
             default:
             }
        }
        sendResult := func(r time.Time) {
            for {
                select {
                case <-pulse:
                    sendPulse()
                case results <- r:
                    return
                }
            }
        }

        for i := 0; i<2; i++ { ❶
            select {
            case <-done:
                return
            case <-pulse:
                sendPulse()
            case r := <-workGen:
                sendResult(r)
            }
        }
    }()
    return heartbeat, results
}

done := make(chan interface{})
time.AfterFunc(10*time.Second, func() { close(done) })

const timeout = 2 * time.Second
heartbeat, results := doWork(done, timeout/2)
for {
    select {
    case _, ok := <-heartbeat:
        if ok == false {
             return
        }
        fmt.Println("pulse")
    case r, ok := <-results:
        if ok == false {
             return
        }
```

```
            fmt.Printf("results %v\n", r)
        case <-time.After(timeout):
            fmt.Println("worker goroutine is not healthy!")
            return
        }
    }
```

❶ 这是我们模拟的问题。所以它不是无限循环的，不需要我们手动停止，就像前面的例子一样，我们只会循环两次。

运行此代码得到下面的结果：

```
pulse
pulse
worker goroutine is not healthy!
```

非常好！在两秒之内，我们的系统意识到我们的 goroutine 有一些不妥之处，中断了 for-select 循环。通过使用心跳，我们已经成功地避免了死锁，并且我们不需要依赖更长的超时时间来保持确定性。我们将在本章后面“治愈异常的 goroutine”中进一步讨论如何进一步采用这个概念。

另外请注意，心跳也会有反作用：虽然它让我们知道，长时间运行的 goroutine 依然正常工作着，但是这需要一点时间运行，计算出值并发送给 channel。

现在，让我们暂时放下间隔心跳，来看看在一个工作单元开始时发出的心跳。它对于测试来说非常有效。以下是在每个工作单元开始之前发送的例子：

```
doWork := func(done <-chan interface{},) (<-chan interface{}, <-chan int){
    heartbeatStream := make(chan interface{}, 1) ❶
    workStream := make(chan int)
    go func(){
        defer close(heartbeatStream)
        defer close(workStream)

        for i:=0; i<10; i++ {
             select { ❷
             case heartbeatStream <- struct{}{}:
             default: ❸
             }
```

```
                select {
                case <-done:
                        return
                case workStream <- rand.Intn(10):
                }
            }
        }()

        return heartbeatStream, workStream
    }

    done := make(chan interface{})
    defer close(done)

    heartbeat, results := doWork(done)
    for {
        select {
        case _, ok := <-heartbeat:
            if ok {
                fmt.Println("pulse")
            } else {
                return
            }
        case r, ok := <-results:
            if ok {
                fmt.Printf("results %v\n", r)
            } else {
                return
            }
        }
    }
```

❶ 在这里，我们创建一个缓冲区大小为 1 的 heartbeat channel 。这确保了即使没有及时接收发送的消息，至少也会发出一个心跳。

❷ 在这里，我们为心跳设置了一个单独的 select 块。我们希望将发送 results 和心跳分开，因为如果接收者没有准备好接收结果，作为替代它将接收到一个心跳，而代表当前结果的值将会丢失。由于我们有默认逻辑，所以这里也没有包含对 done channel 的处理。

❸ 在这里，为了防止没人接收我们的心跳，我们增加了默认逻辑。因为我们的 heartbeat channel 创建时有一个缓冲区那么大，所以如果有人正在监听，但是没有及时收到第一个心跳，接收者仍然可以收到心跳。

运行此代码得到下面的结果：

```
pulse
results 1
pulse
results 7
pulse
results 7
pulse
results 9
pulse
results 1
pulse
results 8
pulse
results 5
pulse
results 0
pulse
results 6
pulse
results 0
```

你可以看到，每个结果都如预期一样伴随着一个心跳。

这种写法真正的亮点在于测试。基于时间间隔的心跳可以以相同的方式使用，但是如果你只关心 goroutine 是否开始了它的工作，这有一种很简单的方式。观察下面的代码：

```
func DoWork(
    done <-chan interface{},
    nums ...int,
) (<-chan interface{}, <-chan int) {
    heartbeat := make(chan interface{}, 1)
    intStream := make(chan int)
    go func() {
        defer close(heartbeat)
        defer close(intStream)

        time.Sleep(2*time.Second) ❶

        for_,n:=rangenums{
             select {
             case heartbeat <- struct{}{}:
             default:
             }
             select {
             case <-done:
                      return
             case intStream <- n:
```

```
            }
        }
    }()

    return heartbeat, intStream
}
```

❶ 在这里，我们模拟在 goroutine 开始工作之前的某种延迟。在实践中，这可能是各种各样的问题，而且无法确定。我曾经见过 CPU 负载过高、磁盘抢占、网络延迟和 goblins 造成的延迟。

DoWork 函数是一个非常简单的生成器，它将我们传入的数字转发到它返回的 channel 中。让我们为这个函数写个测试。下面是一个不那么好的测试样例：

```
func TestDoWork_GeneratesAllNumbers(t *testing.T) {
    done := make(chan interface{})
    defer close(done)

    intSlice := []int{0, 1, 2, 3, 5}
    _, results := DoWork(done, intSlice…)

    for i, expected := range intSlice {
        select {
        case r := <-results:
            if r != expected {
                t.Errorf(
                    "index %v: expected %v, but received %v,",
                    i,
                    expected,
                    r,
                )
            }
        case <-time.After(1 * time.Second): ❶
            t.Fatal("test timed out")
        }
    }
}
```

❶ 在这里我们设置一个合理的超时时间，避免 goroutine 陷入死锁。

运行此代码得到下面的结果：

```
go test ./bad_concurrent_test.go
```

```
--- FAIL: TestDoWork_GeneratesAllNumbers (1.00s)
    bad_concurrent_test.go:46: test timed out
FAIL
FAIL    command-line-arguments  1.002s
```

这个测试不够好，因为它是非确定性的。虽然在我们的示例函数中，已经确保这个测试总是会失败，但如果我要删除这个 time.Sleep 的话，情况可能变得更糟：这个测试有时会通过，有时则会失败。

我们前面提到过，一些外部的因素会导致 goroutine 花费更长的时间来进行第一次迭代。无论 goroutine 在调度中是否是第一位执行的，这都是一个令人担忧的问题。关键是我们无法保证 goroutine 的第一个迭代是否会在超时时间结束之前执行，我们思考一下这个概率：这个超时有多大可能是有意义的？我们可以增加超时时间，但这意味着将需要很长时间才知道执行失败，从而减慢我们的测试过程。

这会产生一些非常可怕的后果。我们慢慢开始不相信测试，然后开始忽略测试，之前的努力将一点点被瓦解。

幸运的是，利用心跳可以很轻易的解决这个问题。这是一个确定性的测试：

```
func TestDoWork_GeneratesAllNumbers(t *testing.T) {
    done := make(chan interface{})
    defer close(done)

    intSlice := []int{0, 1, 2, 3, 5}
    heartbeat, results := DoWork(done, intSlice…)

    <-heartbeat ❶

    i:=0
    for r := range results {
        if expected := intSlice[i]; r != expected {
            t.Errorf("index %v: expected %v, but received %v,", i, expected, r)
        }
        i++
    }
}
```

❶ 在这里，我们等待 goroutine 开始处理迭代的信号。

运行测试将产生以下输出：

```
ok  command-line-arguments    2.002s
```

由于有了心跳，我们可以安全的编写测试而不需要加入超时机制。除此之外还有一个风险，就是我们的一个迭代要花费大量的时间。如果这对我们很重要，那我们可以利用更安全的间隔心跳，从而实现完美的安全性。

下面是一个使用心跳进行测试的例子：

```
func DoWork(
    done <-chan interface{},
    pulseInterval time.Duration,
    nums ...int,
) (<-chan interface{}, <-chan int) {
    heartbeat := make(chan interface{}, 1)
    intStream := make(chan int)
    go func() {
        defer close(heartbeat)
        defer close(intStream)

        time.Sleep(2*time.Second)

        pulse := time.Tick(pulseInterval)
        numLoop: ❷
        for _,n := range nums {
             for { ❶
                      select {
                      case <-done:
                               return
                      case <-pulse:
                               select {
                               case heartbeat <- struct{}{}:
                               default:
                               }
                      case intStream <- n:
                               continue numLoop ❸
                      }
             }
        }
    }()

    return heartbeat, intStream
}

func TestDoWork_GeneratesAllNumbers(t *testing.T) {
    done := make(chan interface{})
    defer close(done)
```

```
    intSlice := []int{0, 1, 2, 3, 5}
    const timeout = 2*time.Second
    heartbeat, results := DoWork(done, timeout/2, intSlice…)

    <-heartbeat ❹

    i:=0
    for {
        select {
        case r, ok := <-results:
            if ok == false {
                return
            } else if expected := intSlice[i]; r != expected {
                t.Errorf(
                        "index %v: expected %v, but received %v,",
                        i,
                        expected,
                        r,
                )
            }
            i++
        case <-heartbeat: ❺
        case <-time.After(timeout):
            t.Fatal("test timed out")
        }
    }
}
```

❶ 我们需要两个循环：一个循环遍历数列，内部循环持续执行，直到 intStream 中的数字成功发送。

❷ 使用一个标签来使简化内部循环。

❸ 跳回 numLoop 标签继续执行外部循环。

❹ 等待第一次心跳到达，来确认 goroutine 已经进入了循环。

❺ 接收心跳，以防止超时。

运行此代码得到下面的结果：

```
ok  command-line-arguments          3.002s
```

你应该注意到了，这个版本的测试不太清晰，我们测试的逻辑有点混乱。因此，如果你确信 goroutine 的循环一旦执行就不会停止，那么我建议只阻塞第一次心跳，然后进入一个简单的 range 语句。你可以编写单独的测试，专门测试未能关闭 channel，耗时太长的迭代，以及其他与时间有关的问题。

在编写并发代码时，心跳并不是必需的，但这部分展示了一些它的实际效果。对于任何长时间运行或需要被测试的 goroutine，我强烈推荐这种模式。

复制请求

对于某些应用来说，尽可能快地接收响应是重中之重。例如，程序正在处理用户的 HTTP 请求，或者检索一个数据块。在这些情况下，你可以进行权衡：你可以将请求分发到多个处理程序（无论是 goroutine，进程，还是服务器），其中一个将比其他处理程序返回更快，你可以立即返回结果。缺点是为了维持多个实例的运行，你将不得不消耗更多的资源。

如果这种复制是在内存中进行的，消耗则没有那么大，但是如果多个处理程序需要多个进程，服务器甚至是数据中心，那可能会变得相当昂贵。所以你需要决定这么做是否值得。

我们来看看如何在单个进程中制造复制请求。我们将使用多个 goroutine 作为处理程序，并且 goroutine 将随机休眠一段时间以模拟不同的负载，休眠时间在 1 到 6 纳秒之间。这将使我们处理程序在不同的时间返回结果，并且我们可以看到复制请求如何更快的返回结果。

下面是一个在 10 个处理程序上复制模拟请求的例子：

```
doWork := func(
    done <-chan interface{},
    id int,
    wg *sync.WaitGroup,
    result chan<- int,
){
    started := time.Now()
    defer wg.Done()

     // 模拟随机负载
    simulatedLoadTime := time.Duration(1+rand.Intn(5))*time.Second
    select {
    case <-done:
    case <-time.After(simulatedLoadTime):
    }
```


```
        select {
        case <-done:
        case result <- id:
        }

        took := time.Since(started)
        // 显示处理程序需要多长时间
        if took < simulatedLoadTime {
            took = simulatedLoadTime
        }
        fmt.Printf("%v took %v\n", id, took)
    }

    done := make(chan interface{})
    result := make(chan int)

    var wg sync.WaitGroup
    wg.Add(10)

    for i := 0; i<10; i++ { ❶
        go doWork(done, i, &wg, result)
    }

    firstReturned := <-result ❷
    close(done) ❸
    wg.Wait()

    fmt.Printf("Received an answer from #%v\n", firstReturned)
```

❶ 在这里，我们启动 10 个处理程序来处理请求。

❷ 在这一行获得处理程序组的第一个返回值。

❸ 在这里，我们取消其余的处理程序。以保证他们不会继续做多余的工作。

运行此代码得到下面的结果：

```
8 took 1.000211046s
4 took 3s
9 took 2s
1 took 1.000568933s
7 took 2s
3 took 1.000590992s
5 took 5s
0 took 3s
6 took 4s
2 took 2s
Received an answer from #8
```


从上面的日志看出，这次是第 8 个处理程序返回的最快。在输出中我们将显示每个处理程序所花费的时间，以便你可以了解这样处理可以节省多少时间。想象一下，如果你只运行一个处理程序，而恰巧是第 5 个处理程序，那请求将不得不等待 5s 才能被处理，而不是刚刚超过 1s 就能被处理完。

有一点需要注意，所有的处理程序都应该是尽可能等价的，有相同的机会处理请求。或者说，你不能从一个无法处理请求的程序那获得最快的响应。就像我说的，每个处理程序都应该有相同的资源。

同一问题的不通特征是大致相似的。你的处理程序越相似，那出现意外的概率就越小。在增加副本时，你应该尽可能复制：你应该只将这样的请求复制到具有不同运行时条件的处理程序。不同的进程，机器，存储路径以及不同的数据源。

虽然建立和维护这样一套系统有很大的代价，但如果你追求的是响应速度，那这是一种非常有价值的架构。另外，这种方式天然地提供了容错和可扩展性。

速率限制

如果你曾经使用过一个 API 服务，那么你可能了解过速率限制，它限制了某种资源在某段时间内被访问的次数。资源可以是任何东西：API 连接，磁盘读写，网络包，异常。

你有没有想过，为什么要在服务中加入速率限制？为什么不允许不受限制地访问系统？通常对系统进行限速，可以避免你的系统被攻击。如果恶意用户可以在资源允许的情况下频繁访问系统，他们可以做各种各样的事情。

例如，他们可以使用日志或有效请求来占满服务器的磁盘。如果你错误地配置了日志转发，它们甚至可以在执行一些恶意的操作后发出足够的请求，将所有的恶意操作记录从日志中挤出，转发到 `/dev/null` 中。他们可能试图暴

力访问资源，或者他们仅仅是执行分布式拒绝服务攻击（DDoS）。重点是：如果你不对系统进行限速，则无法轻松保护它。

可能被恶意利用不是唯一的原因。在分布式系统中，即使是合法用户，如果他们正在以足够大的量级执行操作或者正在运行的代码是异常的，也可能会降低系统的可用性，从而对其他用户的使用造成影响。这甚至可能导致我们之前讨论的死亡螺旋。从产品的角度来看，这非常可怕！通常情况下，你希望向用户提供某种类型的性能保证，这些性能可以保持一致。如果任意一个用户都可以影响这个平衡，那无疑是非常糟糕的。通常情况下用户对系统的访问应当被沙盒化，既不会影响其他用户的活动，也不会受到其他用户的影响。如果你打破了这种思维模式，会使用户感觉你的系统设计不够好，可能会使用户生气或离开。

即使只有一个用户，限速也是有利的。很多时候，即使是开发完成的系统，通常在普通用例下运行良好，但是在不同情况下可能开始有不同的表现。在复杂的分布式系统中，这种影响可能通过系统级联放大，产生意想不到的后果。也许在高负载下，你开始丢包，这导致你的分布式数据库无法仲裁，并停止接受写入，继而使你现有的请求失败，以至于……你可以想象一个简单地问题如何演变成一场灾难。在这些情况下，系统对自己进行一种 DDoS 攻击并不是闻所未闻的！

一个真实的案例

我曾经使用过一个分布式系统，它通过启动新的进程来并行扩展（这可以使它水平扩展到多台机器）。每个进程会新建一个数据库连接，读一些数据，做一些计算。在当时，我们使用这种方式成功扩大了系统规模，满足了客户的需求。但是，一段时间之后，系统利用率增长到了某个点，数据库读取就会超时了。

我们的数据库管理员花了很多时间去检查日志，尝试找出异常的原因。最后他们发现，由于系统上没有任何速率限制，进程互相之间产生了拥挤。

> 由于不同的进程尝试从磁盘的不同部分读取数据，使得磁盘使用率增长到了100%，而且居高不下。这导致了某种恶性循环，系统不断重试，超时。任务无法完成，也无法停止。
>
> 随后我们设计了一个系统来限制数据库的可连接数，限制每秒可读的连接，问题就消失了。虽然客户的操作不得不等待更长的时间才能完成，但至少操作是可以完成的。之后我们有足够的时间进行适当的容量规划，以一种有机的方式扩大系统的容量。

速率限制允许你讲系统的性能和稳定性平衡在可控范围内。如果你需要扩大这些限制，你可以在大量测试和等待后，以可控的方式进行拓展。

在你对系统的访问收费的情况下，速率限制可以使你与客户保持良好的关系。你可以让用户在严格受限的速率下试用这个系统。Google 在其云服务上通过这种方式获得了巨大的成功。

在成为付费用户之后，速率限制甚至可以保护用户。由于大部分时间访问行为是程序化的，因此很容易产生一个失控的 bug 去访问付费系统。这可能会产生巨额的账单，并使双方处于一个非常尴尬境地：服务提供者是否应该承担这些成本，免除无意义访问的费用，或者是用户被迫支付账单，但这可能会对双方的关系造成不可逆的影响？

通常限速是从服务提供者的角度考虑的，不过也可以被用户利用。如果我只是想理解如何使用服务的 API，那么将速率限制在一个很小的范围，会使人更有安全感，因为我知道这不会造成什么危害。

希望我已经给出了足够的理由来说服你，即使你设定了你认为永远不会达到的限制，限制率也是好的。他们很容易创建，他们解决了很多问题，很难合理解释为什么不使用它们。

那我们怎么在 Go 语言中进行限速呢？

大多数的限速是基于令牌桶算法的。这很容易理解，而且相对容易实现。我们来看看它背后的理论。

如果要访问资源，你必须拥有资源的访问令牌，没有令牌的请求会被拒绝。现在假设这些令牌存储在一个等待被检索使用的桶中。桶的深度为 d，表示一个桶可以容纳 d 个访问令牌。例如，存储桶深度为五，则可以存放五个令牌。

每当你需要访问资源时，都会在桶中删除一个令牌。如果你的存储桶包含五个令牌，前五次访问没有问题，操作正常进行；但是在第六次尝试时，就没有访问令牌可用。你的请求必须排队等待，直到令牌可用，或者被拒绝操作。

下面是一个时间表，使用这种可视化的方式，可以帮助你理解。`time` 表示以秒为单位的时间增量，`bucket` 表示桶中请求令牌的数量，`request` 列中的 `tok` 表示成功请求（在下面所有的时间表中，我们会假设这些请求是瞬时的，以简化描述）。

time	bucket	Request
0	5	tok
0	4	tok
0	3	tok
0	2	tok
0	1	tok
0	0	
1	0	

正如表中所示，第一秒前发出的所有请求都成功响应，之后因为没有可用的令牌，请求被阻塞住。

到目前为止，这还是很容易理解的。那如何补充令牌；我们总是能获得一个新的吗？在令牌桶算法中，我们将 r 定义为向桶中添加令牌的速率。它可以是一纳秒或一分钟。这就是我们通常认为的速率限制：因为我们必须等到新的令牌可用，我们将操作速度限制在这个频率下。

下面这个示例中，令牌桶深度为 1 ，速率为 1 令牌 / 秒：

time	bucket	Request
0	1	
0	0	tok
1	0	
2	1	
2	0	tok
3	0	
4	1	
4	0	tok

你可以看到我们的请求立刻就有响应，但是我们的请求被限制在每隔一秒一次。我们的限速代码工作的很正常！

现在我们有两个设置项可以修改：有多少个令牌可以立即使用 d，桶的深度，以及它们补充的速度 r。在这两者之间，我们可以平衡突发性和限制整体速率。突发性指的是当存储桶已满时可以进行多少次请求。

下面的示例中，令牌桶深度为 5 ，速率为 0.5 令牌 / 秒：

time	bucket	Request
0	5	
0	4	tok
0	3	tok
0	2	tok
0	1	tok
0	0	tok
1	0(0.5)	
2	1	
2	0	tok
3	0(0.5)	
4	1	
4	0	tok

在这个示例中，我们能够立刻完成 5 个请求，在这之后，我们每两秒只能完成一个请求。最密集的请求发生在最开始的时候。

请注意，用户可能不会在短时间内消耗掉整个令牌桶的令牌。桶的深度只能控制桶的容量。下面示例中用户先集中发出了两个请求，然后四秒钟之后集中发出 5 个请求：

time	bucket	Request
0	5	
0	4	tok
0	3	tok
1	3	
2	4	
3	5	
4	5	
5	4	tok
5	3	tok
5	2	tok
5	1	tok
5	0	tok

只要用户拥有可用的令牌，集中的请求可能会使用户突破系统的可用范围。有些用户会间歇性访问系统，但是又想要尽可能快的获得结果，就会出现突发性的事件。你只需要确保你的系统能够同时处理所有用户的突发请求，或者在统计上不会有太多的用户同时进行突发的访问。无论采取哪种方式，速率限制都可以使你控制住潜在的风险。

让我们来实现一下这个算法，看看在 Go 语言程序中加入令牌桶算法时会存在那些问题。

让我们假设这有一个可访问的 API，然后提供一个 Go 语言客户端来调用它。该 API 有两个接口：一个用于读取文件，另一个用于将域名解析为 IP 地址。为了简单起见，我们将忽略所有参数并返回实际访问服务所需的值。下面是我们的客户端代码：

```
func Open() *APIConnection {
    return &APIConnection{}
}

type APIConnection struct {}
func (a *APIConnection) ReadFile(ctx context.Context) error {
    // 假装我们在这里运行
    return nil
}

func (a *APIConnection) ResolveAddress(ctx context.Context) error {
    // 假装我们在这里运行
    return nil
}
```

理论上讲我们的请求是经过网络的，所以我们将 `context.Context` 作为第一个参数，以便我们需要取消请求或者将值传递给服务器。非常标准的一段代码。

现在我们将创建一个简单的驱动程序来访问这些 API。这个程序需要读取 10 个文件并解析成 10 个地址，但文件和地址之间没有直接关系，所以驱动程序需要调用那些 API，其中会有一些相互的并发调用。后面我们可以用这个压测 `APIClient`，对我们的限速做一个测试。

```
func main() {
    defer log.Printf( “Done.")
```

```
    log.SetOutput(os.Stdout)
    log.SetFlags(log.Ltime | log.LUTC)

    apiConnection := Open()
    var wg sync.WaitGroup
    wg.Add(20)

    for i:=0; i<10; i++{
        go func() {
            defer wg.Done()
            err := apiConnection.ReadFile(context.Background())
            if err != nil {
                    log.Printf("cannot ReadFile: %v", err)
            }
            log.Printf("ReadFile")
        }()
    }

    for i:=0; i<10; i++ {
        go func() {
            defer wg.Done()
            err := apiConnection.ResolveAddress(context.Background())
            if err != nil {
                    log.Printf("cannot ResolveAddress: %v", err)
            }
            log.Printf("ResolveAddress")
        }()
    }

    wg.Wait()
}
```

运行此代码得到下面的结果：

```
20:13:13 ResolveAddress
20:13:13 ReadFile
20:13:13 ResolveAddress
20:13:13 ReadFile
20:13:13 ReadFile
20:13:13 ReadFile
20:13:13 ReadFile
20:13:13 ResolveAddress
20:13:13 ResolveAddress
20:13:13 ReadFile
20:13:13 ResolveAddress
20:13:13 ResolveAddress
20:13:13 ResolveAddress
20:13:13 ResolveAddress
20:13:13 ResolveAddress
20:13:13 ResolveAddress
```

```
20:13:13 ReadFile
20:13:13 ReadFile
20:13:13 ReadFile
20:13:13 ReadFile
20:13:13 Done.
```

我们可以看到，所有的 API 请求几乎是同时进行的。我们没有进行限速，所以我们的客户端可以自由访问系统。现在我要告诉你一个不幸的消息，我们的驱动程序中可能存在一个可能导致无限循环的 bug。如果没有限速，那一会送账单的快递就要敲门了。

下面我们引入一个限速器。我打算把限速放在 APIConnection 中，但通常限速器会在服务器上运行，这可以防止用户轻易的绕过它。在生产中，客户端也会有限速，以免用户被不必要的拒绝信息干扰，不过这只是锦上添花。不过就我们的目的而言，将限速放在客户端的可以使整个过程变得更简单。

我们将分析一个实例，它使用了 golang.org/x/time/rate 包中的令牌桶限速器实现。我使用这个包是因为它跟我能得到的标准库差不多。当然还有一些其他的软件包，它们可以实现相同的功能，而且有更多在生产中都有意义的功能。而 golang.org/x/time/rate 包很简单，足够我们现在使用。

首先我们会使用这个软件包中两个部分，分别是 Limit 类型和 NewLimiter 函数，在这里定义：

```
//Limit 定义了某些事件的最大频率。
//Limit 表示为每秒事件数。
//zero Limit 不允许发生任何事件
type Limit float64

// NewLimiter 返回一个新的 Limit，
// 它允许事件速率为 r，并允许最大数为 b 的 token
func NewLimiter(r Limit, b int) *Limiter
```

在 NewLimiter 中，我们看到两个熟悉的参数：r 和 b。r 是我们之前说的速率，b 是桶深度。

rates 包也定义了一个辅助方法 Every，用来辅助我们将 time.Duration 转换为一个 Limit：

```
// 每一个都将事件之间的最小时间间隔转换为一个 Limit
func Every(interval time.Duration) Limit
```

Every 函数是有意义的，但是我更想针对每次操作的间隔时间进行测量，而不是请求的间隔长度。我们可以这样写：

```
rate.Limit(events/timePeriod.Seconds())
```

但是我不想每次都手动输入，而且如果时间间隔为零 Every 函数会进行一些特殊的处理，将返回 rate.Inf ，表示没有限制。所以我们将用 Every 函数包装我们的逻辑：

```
func Per(eventCount int, duration time.Duration) rate.Limit {
    return rate.Every(duration/time.Duration(eventCount))
}
```

在创建 rate.Limiter 之后，我们将使用它来阻塞我们的请求，直到获得访问令牌。我们可以用 Wait 方法来做到这一点，它只是调用了一下 WaitN，并将参数设为 1：

```
// Wait 是 WaitN(ctx, 1) 的缩写
func (lim *Limiter) Wait(ctx context.Context)

//WaitN 会执行直到有 n 个事件发生
// 如果 n 超过 Limiter 的突发大小，context 被取消，或者预期等待时间超过
//context 的 deadline，它会返回一个错误
func (lim *Limiter) WaitN(ctx context.Context, n int) (err error)
```

现在我们应该拥有了限制 API 调用速率所有应有的要素。让我们修改一下 APIConnection，然后测试一下！

```
func Open() *APIConnection {
    return &APIConnection{
        rateLimiter: rate.NewLimiter(rate.Limit(1), 1), ❶
    }
}

type APIConnection struct {
    rateLimiter *rate.Limiter
}
```

```
func (a *APIConnection) ReadFile(ctx context.Context) error {
    if err := a.rateLimiter.Wait(ctx); err != nil { ❷
        return err
    }
    // 假设我们在这里执行一些逻辑
    return nil
}

func (a *APIConnection) ResolveAddress(ctx context.Context) error {
    if err := a.rateLimiter.Wait(ctx); err != nil { ❷
        return err
    }
    // 假设我们在这里执行一些逻辑
    return nil
}
```

❶ 在这里，我们将所有 API 连接的速率限制设置为每秒一次。

❷ 我们等待限速器有足够的令牌来完成我们的请求。

运行此代码得到下面的结果：

```
22:08:30 ResolveAddress
22:08:31 ReadFile
22:08:32 ReadFile
22:08:33 ReadFile
22:08:34 ResolveAddress
22:08:35 ResolveAddress
22:08:36 ResolveAddress
22:08:37 ResolveAddress
22:08:38 ResolveAddress
22:08:39 ReadFile
22:08:40 ResolveAddress
22:08:41 ResolveAddress
22:08:42 ResolveAddress
22:08:43 ResolveAddress
22:08:44 ReadFile
22:08:45 ReadFile
22:08:46 ReadFile
22:08:47 ReadFile
22:08:48 ReadFile
22:08:49 ReadFile
22:08:49 Done.
```

你可以看到，在我们同时处理所有的 API 请求之前，已经完成了一次操作。看起来我们的限速器已经在正常工作了！

这就有了一个非常基本的限速，但在生产中，我们可能会想要一些更完备的东西。我们可能会想要建立多层次的限制：用细粒度的控制来限制每秒的请求，用粗粒度的控制来限制每分钟、每小时或每天的请求。

在某些情况下，可以用单一的限速器来做到这一点。然而并不能适应所有情况，而且试图从语义上将单位时间的限制转换为单一层次的限制，过程中会丢失大量信息。由于这些原因，我发现把不同粒度的限速器独立，然后将它们组合成一个限速器组来管理你的调用会更容易一些。为此，我创建了一个简单的聚合限速器 `multiLimiter` 。定义如下：

```
type RateLimiter interface { ❶
    Wait(context.Context) error
    Limit() rate.Limit
}

func MultiLimiter(limiters ...RateLimiter) *multiLimiter {
    byLimit := func(i, j int) bool {
        return limiters[i].Limit() < limiters[j].Limit()
    }
    sort.Slice(limiters, byLimit) ❷
    return &multiLimiter{limiters: limiters}
}

type multiLimiter struct {
    limiters []RateLimiter
}

func (l *multiLimiter) Wait(ctx context.Context) error {
    for _, l := range l.limiters {
        if err := l.Wait(ctx); err != nil {
            return err
        }
    }
    return nil
}

func (l *multiLimiter) Limit() rate.Limit {
    return l.limiters[0].Limit() ❸
}
```

❶ 在这里我们定义了一个 `RateLimiter` 接口，使 `MultiLimiter` 可以递归地定义其他的 `MultiLimiter` 实例。

❷ 这里我们实现一个优化，并根据每个 RateLimiter 的 Limit() 进行排序。

❸ 因为我们在 multiLimiter 实例化时对子 RateLimiter 实例进行了排序，所以我们可以直接返回限制最多的限制器，这将是切片 (slice) 中的第一个元素。

Wait 方法会遍历所有的子限速器，并调用 Wait。这些调用可能会阻塞也可能不会阻塞，但无论如何我们都需要通知每个限速器，所以我们可以设法减少一些令牌桶。通过在为所有限速器设置统一的阻塞可以做到这一点，我们需要确保阻塞的时间是可能阻塞的最长时间。这时因为，如果我们阻塞的时间是最长等待时间的子集，那再碰到那个最长的 Wait 时，还要重新计算剩余时间。而又因为，在前面的 Wait 处进行阻塞，后面的 Wait 会持续填充令牌桶，所以任何 Wait 在第一次阻塞过后一定会被立即返回。

现在我们有了可以进行多重限速的限速方法，让我们借此机会增加一些细节。我们将重新定义我们的 APIConnection 加上每秒钟和每分钟的限制：

```
func Open() *APIConnection {
    secondLimit := rate.NewLimiter(Per(2, time.Second), 1) ❶
    minuteLimit := rate.NewLimiter(Per(10, time.Minute), 10) ❷
    return &APIConnection{
        rateLimiter: MultiLimiter(secondLimit, minuteLimit), ❸
    }
}

type APIConnection struct {
    rateLimiter RateLimiter
}

func (a *APIConnection) ReadFile(ctx context.Context) error {
    if err := a.rateLimiter.Wait(ctx); err != nil {
        return err
    }
    // 假设我们在这里执行一些逻辑
    return nil
}

func (a *APIConnection) ResolveAddress(ctx context.Context) error {
    if err := a.rateLimiter.Wait(ctx); err != nil {
        return err
```

```
        }
        // 假设我们在这里执行一些逻辑
        return nil
    }
```

❶ 在这里，我们定义每秒钟的限制，避免突发请求。

❷ 在这里，我们定义每分钟的限制，为用户提供初始池。每秒的限制将确保我们的系统不会因突发请求而超载。

❸ 然后我们组合这两个限制，并将其设置为 APIConnection 的主限速器。

运行此代码得到下面的结果：

```
22:46:10 ResolveAddress
22:46:10 ReadFile
22:46:11 ReadFile
22:46:11 ReadFile
22:46:12 ReadFile
22:46:12 ReadFile
22:46:13 ReadFile
22:46:13 ReadFile
22:46:14 ReadFile
22:46:14 ReadFile
22:46:16 ResolveAddress
22:46:22 ResolveAddress
22:46:28 ReadFile
22:46:34 ResolveAddress
22:46:40 ResolveAddress
22:46:46 ResolveAddress
22:46:52 ResolveAddress
22:46:58 ResolveAddress
22:47:04 ResolveAddress
22:47:10 ResolveAddress
22:47:10 Done.
```

从日志中我们可以看到，客户端每秒发出两个请求，直到第 11 个请求，我们开始每 6 秒钟发出一次请求。这是因为我们耗尽了分钟级限速器的可用令牌，所以限制了请求速度。

为什么第 11 个请求只延时了 2 秒，而不是像之后的请求那样隔了 6 秒，这可能有点违反直觉。原因是这样的，虽然我们将 API 请求限制为 10 次 / 分钟，但是在一分钟内令牌增加是一个递增的过程。当我们第 11 个请求到达是，我们的分钟级限速器已经积累了一个令牌。

通过定义这样的限制，我们可以清晰的表达粗粒度限制，同时仍然可以进行精细化控制。

这项技术也使我们开始考虑除时间以外的维度。当你对一个系统做限制的时候，你可能不只限制一个维度的东西。你可能对 API 请求的数量有一些限制，同时也可能对其他资源（如磁盘访问，网络访问等）有限制。让我们稍微扩展一下我们的例子，增加磁盘和网络的限制：

```
func Open() *APIConnection {
    return &APIConnection{
    apiLimit: MultiLimiter( ❶
        rate.NewLimiter(Per(2, time.Second), 2),
        rate.NewLimiter(Per(10, time.Minute), 10),
    ),
    diskLimit: MultiLimiter( ❷
        rate.NewLimiter(rate.Limit(1), 1),
    ),
    networkLimit: MultiLimiter( ❸
        rate.NewLimiter(Per(3, time.Second), 3),
    ),
    }
}

type APIConnection struct {
    networkLimit,
    diskLimit,
    apiLimit RateLimiter
}

func (a *APIConnection) ReadFile(ctx context.Context) error {
    err := MultiLimiter(a.apiLimit, a.diskLimit).Wait(ctx) ❹
    if err != nil {
        return err
    }
    // 假设我们在这里执行一些逻辑
    return nil
}

func (a *APIConnection) ResolveAddress(ctx context.Context) error {
    err := MultiLimiter(a.apiLimit, a.networkLimit).Wait(ctx) ❺
    if err != nil {
        return err
    }
    // 假设我们在这里执行一些逻辑
    return nil
}
```

❶ 在这里，我们为 API 调用设置了一个限速器。每秒请求数和每分钟请求数都有限制。

❷ 在这里，我们为磁盘读取设置一个限速器。我们将其限制为每秒只能读取一次。

❸ 对于网络，我们设置了每秒三个请求的限制。

❹ 当我们读取文件时，我们融合 API 限速器和磁盘限速器的限制。

❺ 当我们需要网络访问时，我们融合 API 限速器和网络限速器的限制。

运行此代码得到下面的结果：

```
01:40:15 ResolveAddress
01:40:15 ReadFile
01:40:16 ReadFile
01:40:17 ResolveAddress
01:40:17 ResolveAddress
01:40:17 ReadFile
01:40:18 ResolveAddress
01:40:18 ResolveAddress
01:40:19 ResolveAddress
01:40:19 ResolveAddress
01:40:21 ResolveAddress
01:40:27 ResolveAddress
01:40:33 ResolveAddress
01:40:39 ReadFile
01:40:45 ReadFile
01:40:51 ReadFile
01:40:57 ReadFile
01:41:03 ReadFile
01:41:09 ReadFile
01:41:15 ReadFile
01:41:15 Done.
```

我可以在这里画一个时间表来解释每次调用都会发生在哪里，但是这会掩盖真正有意义的东西。相反，我们把重点放在这样一个事情上，即我们能够将逻辑限速器组合成对每个调用都有意义的限制组，并且保证 `APIClient` 正常工作。如果我们观察一下它是如何工作的，我们会注意到涉及网络访问的 API 调用看起来更加规律了，大部分操作在前三分之二的调用中就完成了。这可能和 goroutine 的调度相关，但更有可能是我们的限速器在起作用！

还有一点，rate.Limiter 类型有一些特殊的技巧来进行优化或处理其他用例。我只描述了它的一部分功能，就是阻塞请求直到令牌桶生成一个新的令牌。但如果你只是想使用一下它，那只需要知道它还有其他一些功能即可。

在本节中，我们已经研究了使用速率限制的原因，并实现了一个令牌桶算法，以及研究了如何将令牌桶限速器组合成更大、更复杂的限速器。这应该会让你对速率限制有一个充分的了解，并有助于你在生产中使用它们。

治愈异常的 goroutine

在长期运行的后台程序中，经常会有一些长时间运行的 goroutine。这些 goroutine 经常处于阻塞状态，等待数据以某种方式到达，然后唤醒它们，进行一些处理，再返回一些数据。有时候，这些 goroutine 依赖于一些控制不太好的资源。也许一个 goroutine 需要从接收到的请求中提取数据，或者它正在监听一个临时文件。问题在于，如果没有外部干预，一个 goroutine 很容易进入一个不正常的状态，并且无法恢复。抛开这些担忧，你甚至可以说，goroutine 本身不应该关心其如何从一个异常状态回复过来。在一个长期运行的程序中，建立一个机制来监控你的 goroutine 是否处于健康的状态是很用的，当他们变得异常时，就可以尽快重启。我们将这个重启 goroutine 的过程称为“治愈”（Healing）[注2]。

为了治愈 goroutine，我们需要使用心跳模式来检查我们正在监控的 goroutine 是否活跃。心跳的类型取决于你想要监控的内容，但是如果你的 goroutine 有可能会产生活锁，确保心跳包含某些信息，表明该 gouroutine 在正常的工作而不仅仅是活着。在本节中，为了简单起见，我们只会考虑 goroutine 是否活着。

我们把监控 goroutine 的健康这段逻辑称为管理员，它监视一个管理区的 goroutine。如果有 goroutine 变得不健康，管理员将负责重新启动这个管理区

注 2：熟悉 erlang 的人可能会知道这个概念！Erlang 的监控器 (supervisors) 也做了同样的事情。

的 goroutine。为此，需要引用一个可以启动 goroutine 的函数。让我们看看一个管理员应该是什么样的：

```
type startGoroutineFn func(
    done <-chan interface{},
    pulseInterval time.Duration,
) (heartbeat <-chan interface{}) ❶

newSteward := func(
    timeout time.Duration,
    startGoroutine startGoroutineFn,
) startGoroutineFn {
    return func(
        done <-chan interface{},
        pulseInterval time.Duration,
    ) (<-chan interface{}) { ❷
        heartbeat := make(chan interface{})
        go func() {
            defer close(heartbeat)

            var wardDone chan interface{}
            var wardHeartbeat <-chan interface{}
            startWard := func() { ❸
                wardDone = make(chan interface{}) ❹
                wardHeartbeat = startGoroutine(or(wardDone, done),timeout/2)❺
            }
            startWard()
            pulse := time.Tick(pulseInterval)

        monitorLoop:
            for {
                timeoutSignal := time.After(timeout)

                for { ❻
                    select {
                    case <-pulse:
                        select {
                        case heartbeat <- struct{}{}:
                        default:
                        }
                    case <-wardHeartbeat: ❼
                        continue monitorLoop
                    case <-timeoutSignal: ❽
                        log.Println("steward: ward unhealthy; restarting")
                        close(wardDone)
                        startWard()
                        continue monitorLoop
                    case <-done:
```

```
                        return
                    }
                }
            }
        }()

        return heartbeat
    }
}
```

❶ 在这里，我们定义一个可以监控和重启的 goroutine 的信号。我们看到了熟悉的 channel，以及来自心跳模式的 `pulseInterval` 和 `heartbeat`。

❷ 在这一行，我们看到一个管理员监控 goroutine 需要的 `timeout` 变量，还有一个函数 `startGoroutine` 来启动它监控的 goroutine。有趣的是，管理员本身会返回一个 `startGoroutineFn`，表示管理员本身也是可监控的。

❸ 在这里，我们定义一个闭包，它实现了一个统一的方法来启动我们正在监视的 goroutine。

❹ 这是我们创建的一个新的 channel ，如果我们需要发出一个停止的信号，就会通过它传入 goroutine 中。

❺ 在这里，我们启动将要监控的 goroutine。如果管理员被停止了，或者管理员想要停止 goroutine，我们希望这些信息都能传递给管理区里的 goroutine ，所以我们把两个 done channel 用逻辑或包装了一下。我们设定心跳间隔时间是超时时间的一半，我们在本章前面“心跳”中讨论过，这里只是提一下。

❻ 这是我们的内部循环，它确保管理员可以发出自己的心跳。

❼ 在这里，如果我们收到 goroutine 的心跳，将继续执行监控的循环。

❽ 这一行表示，如果我们在暂停期间没有收到管理区里 goroutine 的心跳，我们会要求管理区里的 goroutine 停下来，并启动一个新的 goroutine。然后我们继续监控。

我们的 `for` 循环有点多，但只要你熟悉其中涉及的模式，阅读起来还是相

对简单的。让我们对管理员进行一个测试。如果我们监控一个的行为异常的goroutine会发生什么？让我们来看看：

```
log.SetOutput(os.Stdout)
log.SetFlags(log.Ltime | log.LUTC)

doWork := func(done <-chan interface{}, _ time.Duration) <-chan interface{} {
    log.Println("ward: Hello, I'm irresponsible!")
    go func() {
        <-done ❶
        log.Println("ward: I am halting.")
    }()
    return nil
}
doWorkWithSteward := newSteward(4*time.Second, doWork) ❷

done := make(chan interface{})
time.AfterFunc(9*time.Second, func() { ❸
    log.Println("main: halting steward and ward.")
    close(done)
})

for range doWorkWithSteward(done, 4*time.Second) {} ❹
log.Println("Done")
```

❶ 在这里，我们看到这个goroutine没有做任何事情，只是等待被取消。它也没有发出任何心跳。

❷ 这一行创建了一个函数，为goroutine `doWork`创建一个管理员。我们设置`doWork`的超时时间为4秒钟。

❸ 在这里，我们设置9秒钟之后停止管理员和goroutine，这样我们的测试就会结束。

❹ 最后，我们启动管理员并在其心跳范围内防止我们的测试停止。

这个示例产生以下输出：

```
18:28:07 ward: Hello, I'm irresponsible!
18:28:11 steward: ward unhealthy; restarting
18:28:11 ward: Hello, I'm irresponsible!
18:28:11 ward: I am halting.
18:28:15 steward: ward unhealthy; restarting
18:28:15 ward: Hello, I'm irresponsible!
18:28:15 ward: I am halting.
```

```
18:28:16 main: halting steward and ward.
18:28:16 ward: I am halting.
18:28:16 Done
```

它看起来运行得很好！ 不过我们的管理区有些简单：除了取消和心跳所需要的东西之外，它不接收任何参数，也不返回任何参数。我们怎样才能创造一个可以与管理员配合使用的管理区呢？我们每次都可以改写或者生成一个管理员来适应我们的管理区，但是这样做既麻烦又不必要，相反，我们可以使用闭包。让我们来看一个管理区的例子，它根据离散值列表生成一个整数流：

```
doWorkFn := func(
    done <-chan interface{},
    intList ...int,
) (startGoroutineFn, <-chan interface{}) { ❶
    intChanStream := make(chan (<-chan interface{})) ❷
    intStream := bridge(done, intChanStream)
    doWork := func(
        done <-chan interface{},
        pulseInterval time.Duration,
    ) <-chan interface{} { ❸
        intStream := make(chan interface{}) ❹
        heartbeat := make(chan interface{})
        go func() {
            defer close(intStream)
            select {
            case intChanStream <- intStream: ❺
            case <-done:
                return
            }

            pulse := time.Tick(pulseInterval)

            for {
                valueLoop:
                for _, intVal := range intList {
                    if intVal < 0 {
                        log.Printf("negative value: %v\n", intVal) ❻
                        return
                    }

                    for {
                        select {
                        case <-pulse:
                            select {
                            case heartbeat <- struct{}{}:
                            default:
                            }
```

```
                        case intStream <- intVal:
                            continue valueLoop
                        case <-done:
                            return
                        }
                    }
                }
            }
        }()
        return heartbeat
    }
    return doWork, intStream
}
```

❶ 在这里，我们填入一些我们管理区所需的参数，并返回我们管理区用来通信的 channel。

❷ 这一行创建了作为桥接模式一部分的 channel。

❸ 在这里，我们创建一个将被管理员启动和监控的闭包。

❹ 这是我们实例化 channel 的地方，我们将利用这些 channel 与管理区中的 goroutine 通信。

❺ 在这里，我们把我们即将用来通信的 channel 通知给 bridge。

❻ 当我们处理到负数时，在这一行打印出一个错误信息，然后从 goroutine 中返回。

你可以看到，由于我们可能会启动多个管理区的副本，所以我们利用 bridge-channel（请参阅第 4 章“桥接 channel 模式”）向 doWork 的消费者提供一个共用的 channel，避免中断。使用这些技术，我们的管理区可以简单地通过组合各种模式变得任意复杂。让我们看个例子：

```
log.SetFlags(log.Ltime | log.LUTC)
log.SetOutput(os.Stdout)

done := make(chan interface{})
defer close(done)

doWork, intStream := doWorkFn(done, 1, 2, -1, 3, 4, 5) ❶
doWorkWithSteward := newSteward(1*time.Millisecond, doWork) ❷
doWorkWithSteward(done, 1*time.Hour) ❸
```

```
for intVal := range take(done, intStream, 6) { ❹
    fmt.Printf("Received: %v\n", intVal)
}
```

❶ 这一行，我们创建管理区的函数，允许它结束我们的可变整数切片，并返回一个用来返回的流。

❷ 在这里，我们创建一个管理员，用来监听 dowork 闭包。因为我们希望能尽快知道失败的信息，所以我们将监听时间间隔设置为一毫秒。

❸ 在这里，我们通知管理员启动管理区并开始监控。

❹ 最后，使用我们开发的一段管道代码，从 intStream 中取出前六个值。

运行此代码得到下面的结果：

```
Received: 1
23:25:33 negative value: -1
Received: 2
23:25:33 steward: ward unhealthy; restarting
Received: 1
23:25:33 negative value: -1
Received: 2
23:25:33 steward: ward unhealthy; restarting
Received: 1
23:25:33 negative value: -1
Received: 2
```

根据收到的值，我们可以看到管理区内的异常状态，我们的管理员会检查管理区，重启管理区。你可能也注意到了，我们只能接收到 1 和 2 这两个值，这是我们管理区重新初始化的表现。当你开发自己的管理区时，如果你的系统对重复值比较敏感，一定要重视这一点。你还可以考虑在管理员中加入一段逻辑，在失败几次后退出。在这种情况下，我们可以简单地通过更新我们在每次迭代中关闭的 intList，来使得我们的生成器有状态。而在此之前：

```
valueLoop:
for _, intVal := range intList {
    // ...
}
```

我们可以这样写：

```
valueLoop:
    for {
        intVal := intList[0]
        intList = intList[1:]
        // ...
    }
```

这将会在管理区重启时节省一些空间，虽然我们会始终停留在无效的负数上，我们的管理区将会一直处于失败状态。

使用这种模式可以使长期运行着的 goroutine 保持健康状态。

小结

在本章中，我们介绍了一些方法来保持系统的稳定和易于理解，因为这些系统所处理的问题领域需要更大的系统，甚至是分布式的。本章还演示了 Go 语言的并发原语如何创建更高层次的抽象程序。如果在语言设计上没有对并发提供额外的优势，这些模式可能会更笨重，也更不健壮。

在最后一章中，我们将探索 Go 语言的一些运行时的内部结构，以帮助你深入理解它的工作方式。我们还将探索一些有用的工具，使开发和调试 Go 语言程序更容易一些。

第 6 章

goroutine 和 Go 语言运行时

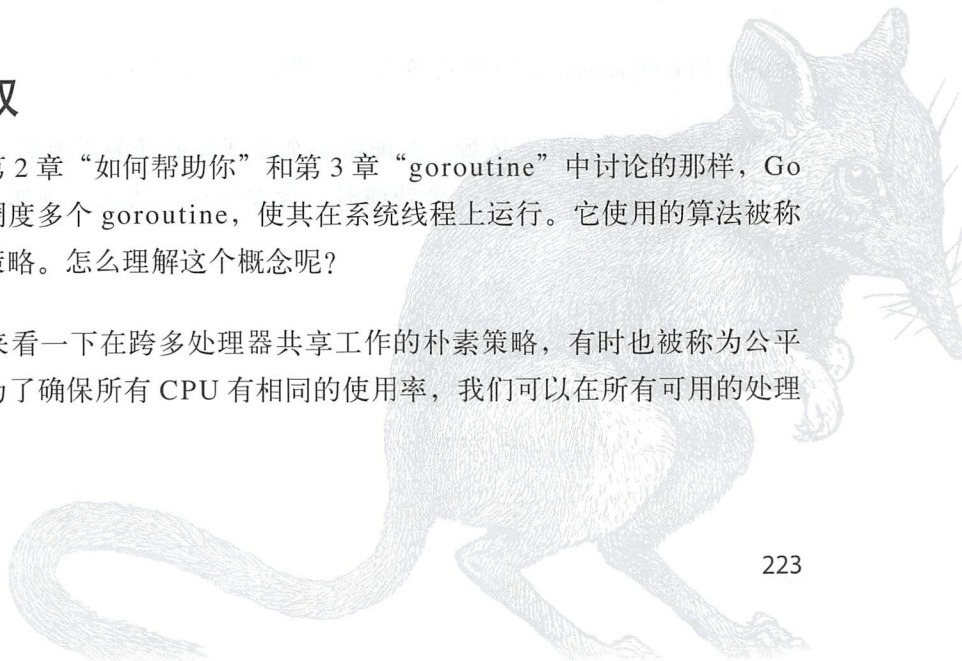
当使用 Go 语言开发时，使用并发是非常有趣的，因为语言本身使这件事变得如此简单！我们很少需要了解运行时是如何将这一切交织在一起的。尽管如此，有些时候这些信息仍然是很有用的，比如第 2 章所讨论的东西都是基于运行时才能做到，所以花一点时间来看看运行时是如何工作地还是很有必要的。如果对它感兴趣会有更多收获。

Go 语言的运行时为你做了很多事情，其中派生（spawning）和管理 goroutine 可能是最有利于你和你的程序的。作为 Go 语言的创造者，Google 对于将计算机科学理论与技术白皮书相结合，有着悠久的历史，所以 Go 语言包含了来自学术界的一些思想并不奇怪。不过令人惊讶的是 goroutine 背后的复杂程度。Go 语言使用了一些强大的思想，使你的程序更加高效，而且还抽象出了这些细节，为开发人员提供了一个非常好的使用体验。

工作窃取

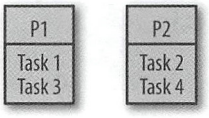
正如我们在第 2 章“如何帮助你”和第 3 章“goroutine”中讨论的那样，Go 语言将为你调度多个 goroutine，使其在系统线程上运行。它使用的算法被称为工作窃取策略。怎么理解这个概念呢？

首先，我们来看一下在跨多处理器共享工作的朴素策略，有时也被称为公平调度策略。为了确保所有 CPU 有相同的使用率，我们可以在所有可用的处理



器之间平均分配负载。想象一下，有 n 个处理器，有 x 个任务要执行。在公平调度策略中，每个处理器都会得到 x/n 个任务：

```
<Schedule Task 1>
<Schedule Task 2>
<Schedule Task 3>
<Schedule Task 4>
```



不过这种方法存在很多问题。如果你还记得第 3 章“goroutine”，Go 语言的并发模型使用的是 fork-join 模型。在一个 fork-join 范例中，任务可能相互依赖，在实际使用过程中，基于朴素策略在处理器上分配任务可能会导致其中一个处理器利用率不足。不仅如此，还可能导致缓存的位置偏差，因为需要调用这些数据的任务跑在其他处理器上。我们来看一个例子，思考下为什么会这样。

想象一个简单的程序，它会产生之前描述的工作分布。如果任务 2 比任务 1 和任务 3 结合需要更长的时间，会发生什么？

Time	P1	P2
	T1	T2
n+a	T3	T2
n+a+b	(idle)	T4

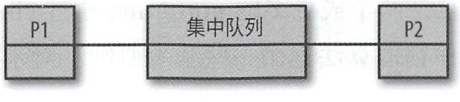
无论 a 和 b 两者的持续时间是多久，处理器 1 一定会闲置。

如果任务之间存在相互依赖，分配给一个处理器的任务需要另一个任务的结果，而该任务分配给了另一个处理器，会发生什么情况？例如，假如任务 1 依赖任务 4 呢？

Time	P1	P2
	T1	T2
n+a	(blocked)	T2
n+a+b	(blocked)	T4
n+a+b+c	T1	(idle)
n+a+b+c+d	T3	(idle)

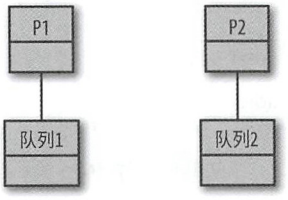
这种情况下，在处理器 1 完全空闲时，任务 2 和 4 进行着计算。处理器 2 在任务 2 中被占用，与此同时处理器 1 在任务 1 上被阻塞，而处理器 1 本可以执行任务 4 以解除自己的阻塞。

好吧，这听起来像是 FIFO 队列可以解决的基础负载平衡问题，那让我们尝试一下：工作任务加入队列中进行调度，我们的处理器在有空闲时的将任务出队，或者阻塞连接。这将是我们看到的第一种工作窃取算法。这是否解决了这个问题呢？



答案是可能会解决。因为它解决了未充分利用的处理器的问题，所以比简单的在处理器之间分配任务要好，但是现在我们引入了一个集中化的队列，所有处理器都必须使用的这个数据结构。正如第 1 章“内存访问同步”中所讨论的，我们知道反复进入和退出临界区的代价非常高。不仅如此，我们缓存偏移问题也会更加突出：现在我们必须使用一个集中的队列，每次想要入队或出队一个任务时继续要将这个队列加载到每个处理器的缓存中。不过，对于粗粒度的操作，这仍是一个行之有效的方法。但是，goroutine 通常不是粗粒度的，所以集中队列可能并不是适合我们的工作调度算法。

我们可以做一些优化，拆分工作队列。我们给每个处理器的一个独立的线程和一个双端队列，如下所示：



好的，我们已经解决了高度竞争下数据结构过于集中的问题，但是缓存局部性和处理器利用率的问题呢？问题还不止这些，如果工作从 P1 开始，所有 fork 出来的任务都放在 P1 的队列中，那么任务什么时候会在 P2 上执行呢？现在，任务在队列之间移动的情况下，上下文切换不会有问题吗？我们来看一下使用分布式队列的工作窃取算法，它的运行是什么样的。

首先我们需要强调下，Go 语言遵循 fork-join 模型进行并发。在 goroutine 开始的时候 fork，join 点是两个或更多的 goroutine 通过 channel 或 sync 包中的类型进行同步时。工作窃取算法遵循一些基本原则。对于给定的线程：

1. 在 fork 点，将任务添加到与线程关联的双端队列的尾部。
2. 如果线程空闲，则选取一个随机的线程，从它关联的双端队列头部窃取工作。
3. 如果在未准备好的 join 点（即与其同步的 goroutine 还没有完成），则将工作从线程的双端队列尾部出栈。
4. 如果线程的双端队列是空的，则：
   a. 暂停加入。
   b. 从随机线程关联的双端队列中窃取工作。

这规则有些抽象，让我们来看一些实际的代码，观察这个算法是怎样运转的。下面的程序在递归计算 Fibonacci 数列：

```
var fib func(n int) <-chan int
fib = func(n int) <-chan int {
    result := make(chan int)
    go func() {
        defer close(result)
        if n <= 2 {
            result <- 1
            return
        }
        result <- <-fib(n-1) + <-fib(n-2)
    }()
    return result
}

fmt.Printf("fib(4) = %d", <-fib(4))
```

让我们来看看当前版本的工作窃取算法是如何在这个 Go 语言程序中运转的。假设让这个程序在我们虚构的机器上运行的，这个机器上有两个单核处理器。我们将在每个处理器上生成一个系统线程，处理器 1 上产生 T1，处理器 2 上产生 T2。当我们运行过一次后，我会将 T1 与 T2 调换一下，以使结构更加严谨。而在实际运行中，这些都不是确定的。

所以让我们开始运行这个程序。初始化时，我们只有一个 goroutine，main goroutine，我们将假设它在处理器 1 上：

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
(main goroutine)			

接下来，我们调用 `fib(4)`。这个 goroutine 将被安排在 T1 的工作队列尾部，并且父 goroutine 将继续运行：

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
(main goroutine)	fib(4)		

此时，根据时机的不同，可能会发生以下两种情况中的一种：T1 或 T2 将盗取调用 `fib(4)` 的 goroutine。对于这个例子，为了更清晰地说明算法，我们假设 T1 成功窃取；然而，有一点很重要，任何线程都可能会成功。

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
(main goroutine)(等待 join)			
fib(4)			

fib(4) 在 T1 上运行，因为加法操作的顺序是从左到右的，所以先添加 fib(3)，之后将 fib(2) 到追加到队列的尾部：

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
(main goroutine)(等待 join)	fib(3)		
fib(4)	fib(2)		

此时，T2 仍然是空闲的，所以它从 T1 的队列头部取出 fib(3)。请注意，fib(2) 是 fib(4) 推入队列的最后一个任务，因此 T1 最有可能需要计算的第一个任务仍然在 T1 上。我们将在后面讨论这为什么非常重要。

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
(main goroutine)(等待 join)	fib(2)	fib(3)	
fib(4)			

与此同时，由于在 fib(3) 和 fib(2) 返回的 channel 上等待着，T1 不足以继续处理 fib(4)。这是我们算法第三步中的 join 点。因此，它会从自己的队列中出栈一个任务，就是 fib(2)：

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
(main goroutine)(等待 join)		fib(3)	
fib(4)(等待 join)			
fib(2)			

这里有些混乱。因为我们没有在递归算法中使用回溯，而是要安排了一个新 goroutine 来计算 fib(2)。刚刚被安排到 T1 上的就是一个独立运行的 goroutine。

上面在 T1 上运行的是 `fib(4)` 调用过程的一部分，即 4-2。下面产生的 goroutine 是 `fib(3)` 调用过程的一部分，即 3-1。这是调用 `fib(3)` 产生的 goroutine：

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
(main goroutine)(等待 join)		fib(3)	fib(2)
fib(4)(等待 join)			fib(1)
fib(2)			

接下来，T1 到达了 Fibonacci 算法的收敛处（$n \leqslant 2$），返回了 1：

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
(main goroutine)(等待 join)		fib(3)	fib(2)
fib(4)(等待 join)			fib(1)
fib(1)			

然后 T2 达到一个 join 点，并从其队列的尾部出栈一个任务：

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
(main goroutine)(等待 join)		fib(3)(等待 join)	fib(2)
fib(4)(等待 join)		fib(1)	
(returns 1)			

现在 T1 又一次处于空闲状态所以它从 T2 的队列中窃取工作。

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
(main goroutine)(等待 join)		fib(3)(等待 join)	
fib(4)(等待 join)		fib(1)	
fib(2)			

T2 再一次到达递归的终结点（$n \leqslant 2$）并返回了 1：

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
(main goroutine)(等待 join)		fib(3)(等待 join)	
fib(4)(等待 join)		(returns 1)	
fib(2)			

接下来，T1 也到达终结点并返回了 1：

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
(main goroutine)(等待 join)		fib(3)(等待 join)	
fib(4)(等待 join)		(returns 1)	
(returns 1)			

T2 对 `fib(3)` 的调用现在有两个已完成的 join 点。也就是说，`fib(2)` 和 `fib(1)` 已经通过他们的 channel 返回了结果，并且 `fib(3)` 产生的两个 goroutine 已经运行结束。它执行加法运算（1 + 1 = 2）并通过它的 channel 返回结果：

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
(main goroutine)(等待 join)		(returns 2)	
fib(4)(等待 join)			

这里又发生了同样地事情：`fib(4)` 调用的 goroutine 有两个 join 点：`fib(3)` 和 `fib(2)`。我们刚刚完成了上一步中的 `fib(3)` 的 join，并且在 T2 的最后一个任务结束时完成了 `fib(2)` 的 join。再次执行加法（2 + 1 = 3），结果通过 `fib(4)` 的 channel 返回：

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
(main goroutine) (等待 join)			
(return 3)			

此时，我们已经实现了 main goroutine 的 join（`<-fib(4)`），然后继续执行 main goroutine。它通过打印来输出结果：

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
(print3)			

现在，让我们来看看这个算法的一些有趣的特性。回想一下，正在执行的线程会在队列的尾部入栈或者（必要时）出栈一个任务。位于队列尾部的任务有这样几个有趣的特性：

这是最有可能完成父进程 join 的任务。

更快地完成join意味着我们的程序性能会更好，在内存中停留的时间更少。

这是最有可能存在于处理器缓存中的任务。

因为这是这个线程在开始当前工作前的最后一个任务。所以当前线程执行需要的信息可能仍然存在于CPU的缓存之中。这意味着缓存的命中率更高！

总的来说，这种任务调度的方式在性能上有很多隐含的好处。

窃取任务还是续体

事实上我们掩盖了一个问题，那就是我们应该让什么样的任务进行排队和窃取。在fork-join模式下，有两种选择：新任务和续体。为了确保你对Go语言中的任务和续体有清晰的了解，让我们重新研究下我们的Fibonacci程序：

```
var fib func(n int) <-chan int
fib = func(n int) <-chan int {
    result := make(chan int)
    go func() { ❶
        defer close(result)
        if n <= 2 {
            result <- 1
            return
        }
        result <- <-fib(n-1) + <-fib(n-2)
    }()
```

```
    return result ❷
}

fmt.Printf("fib(4) = %d", <-fib(4))
```

❶ 在 Go 语言中，goroutine 就是任务。

❷ 在 goroutine 之后的一切都被称为续体。

在之前分布式队列工作窃取算法的编排中，我们将任务或 goroutine 进行排队。由于 goroutine 很好地封装了一个工作体，所以这是一种非常自然的思考方式；然而，这实际上并不是 Go 语言中工作窃取算法的工作原理。Go 语言的工作窃取算法对续体进行入队和窃取。

那么为什么这很重要呢？入队和窃取续体对我们来说有什么作用，入队和窃取任务能不能做到？要开始回答这个问题，我们来看看我们的 join 点。

在我们的算法下，当一个执行线程到达一个 join 点时，该线程必须暂停执行，等待回调以窃取任务。这被称为延时 join，因为它在寻找要执行的任务时，一直在 join 点等待着。任务窃取和续体窃取算法都有延时 join，但延时发生的频率存在显著差异。

设想一下：当创建一个 goroutine 时，你的程序很可能会希望该 goroutine 中的函数被立即执行。同时，其他 goroutine 的续体也有可能想要 join 回那个 goroutine。在 goroutine 执行完成之前，某些续体尝试 join 也并不罕见。对于这些有一条公理，即在调度 goroutine 时，最合理的是立即开始执行这个 goroutine。

现在回想一下线程的属性，这个线程从队列尾部入栈和出栈任务，其他线程从头部出栈任务。如果我们将续体入栈到队列的尾部，则该任务最不可能被线程从头部取出，当我们的 gorouine 需要结束时很可能将它取回，从而避免了延迟。这也使 fork 的任务看起来很像函数调用：线程跳转到执行 goroutine，然后在完成后返回到续体。

让我们看看如何将续体窃取算法应用到我们的 Fibonacci 程序。因为表示续体不像表示任务那样清晰，所以我们将做下约定：

- 当续体在一个工作队列上排队时，我们将把它列为 `cont. of X`。
- 当一个续体被用于执行时，我们做一个隐式转换，将续体转换为 `fib` 的下一个调用。

下面的内容对 Go 语言运行时做了更详细的描述。

我们依旧从 main goroutine 开始：

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
main			

main goroutine 调用 `fib(4)` 并将这个调用的续体追加到 T1 工作队列的尾部：

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
fib(4)	cont. of main		

T2 是空闲的，所以它窃取了 main 的续体：

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
fib(4)		cont. of main	

`fib(4)` 调用了 `fib(3)`，并且立刻就开始运行，同时 T1 将 `fib(4)` 的续体入栈到它的队列尾部：

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
fib(3)	cont. of fib(4)	cont. of main	

当 T2 试图执行 main 的续体时，它是一个等待 join 的点。因此，它从 T1 的队列中窃取了更多的工作。这一次，它得到了 `fib(4)` 续体：

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
fib(3)		cont. of main(等待 join)	
		cont. of fib(4)	

接下来，T1 上的 fib(3) 为 fib(2) 启动了一个 goroutine。fib(3) 的续体被入栈到其工作队列的尾部：

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
fib(2)	cont. of fib(3)	cont. of main	
		cont. of fib(4)	

T2 执行了 T1 未执行的 fib(4) 续体，它又调用了一个新的 fib(2)，fib(4) 的续体又继续进入队列：

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
fib(2)	cont. of fib(3)	cont. of main	cont. of fib(4)
		fib(2)	

接下来，T1 执行 fib(2) 到了递归终结点，返回了 1：

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
(returns 1)	cont. of fib(3)	cont. of main	cont. of fib(4)
		fib(2)	

然后 T2 也到达终结点并返回了 1：

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
(returns 1)	cont of fib(3)	cont. of main (等待 join)	cont. of fib(4)
		(returns 1)	

然后，T1 从它自己的队列中取出了一个任务并开始执行，就是 fib(1)。这里我们看一下 T1 的调用链：fib(3) → fib(2) → fib(1)。这就是我们之前说的续体窃取算法的好处！

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
fib(1)		cont. of main (等待 join)	cont.of fib(4)
		(returns 1)	

然后在 fib(4) 的续体结束时，只有一个 join 被实现：fib(2)。对 fib(3) 的调用仍然由 T1 处理。因为没有工作窃取：

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
fib(1)		cont. of main (等待 join)	
		fib(4)(等待 join)	

T1 现在处于一个续体结束的阶段，fib(3)，它的从 fib(2) 和 fib(1)join 完成。T1 返回 2：

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
(returns 2)		cont. of main (等待 join)	
		(returns 2)	

现在 fib(4)、fib(3) 和 fib(2) 都 join 完成了。T2 能够执行其计算并返回结果 (2+1=3):

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
		cont. of main (等待 join)	
		(returns 3)	

最后，main goroutine 已经 join 完成，它从 fib(4) 收到返回值，然后打印出结果，3：

T1 调用栈	T1 工作队列	T2 调用栈	T2 工作队列
		main (prints 3)	

当我们看过上面那些后，我们简要总结下续体是在 T1 上时如何起作用的。如

果我们看一下这个运行的统计数据（带有连续的窃取），并对比任务窃取相比，就会发现一个更清晰的优势：

统计	续体窃取	任务窃取
步骤	14	15
队列最大长度	2	2
延迟 join	2（所有都在空闲的线程上）	3（所有都在忙碌的线程上）
调用栈最大深度	2	3

这些统计数据似乎很接近，但我们可以推断如果在更大的程序中，我们就会发现续体窃取会带来显著的好处。

让我们在一个线程上执行一下这个程序：

T1 调用栈	T1 工作队列
main	

T1 调用栈	T1 工作队列
fib(4)	main

T1 调用栈	T1 工作队列
fib(3)	main
	cont. of fib(4)

T1 调用栈	T1 工作队列
fib(2)	main
	cont. of fib(4)
	cont. of fib(3)

T1 调用栈	T1 工作队列
(returns 1)	main
	cont. of fib(4)
	cont. of fib(3)

T1 调用栈	T1 工作队列
fib(1)	main
	cont. of fib(4)

T1 调用栈	T1 工作队列
(returns 1)	main
	cont. of fib(4)

T1 调用栈	T1 工作队列
(returns 2)	main
	cont. of fib(4)

T1 调用栈	T1 工作队列
fib(2)	main

T1 调用栈	T1 工作队列
(return 1)	main

T1 调用栈	T1 工作队列
(return 3)	main

T1 调用栈	T1 工作队列
main (print3)	

结果很有趣，在单线程上使用 goroutine 时，它的运行时就像我们在使用的函数一样！这是续体窃取的另一个好处。

综合考虑，从理论上来说，续体窃取被认为是优于任务窃取的，因此，最好是对续体而不是 goroutine 进行排队。从下表可以看出，窃取延续有几个好处：

那么，为什么所有的“工作窃取”算法都不使用“续体窃取”呢？因为续体窃取通常需要编译器的支持。幸运的是，Go 语言有它自己的编译器，并且正是用续体窃取实现的工作窃取算法。没有这种支持的语言通常会使用任务窃取，也就是所谓的“子任务窃取”，窃取作为一个库存在。

	计算续体	子任务
队列长度	有界的	无界的
执行顺序	连续的	次序颠倒的
Join 点	无延迟	有延迟

虽然这个模型更接近于 Go 语言的算法，但它仍然不能代表整体的景象。Go 语言还加入了额外的优化。在我们分析这些之前，让我们先熟悉下在源代码中使用的 Go 语言调度程序的命名方法。

Go 语言的调度器有三个主要的概念：

G

goroutine。

M

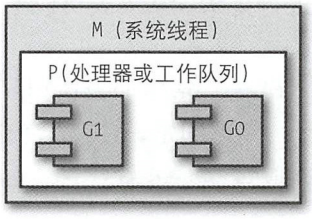
OS 线程（在源代码中也被称为机器）。

P

上下文（在源代码中也被称为处理器）。

在我们关于工作窃取的讨论中，*M* 等于 *T*，而 *P* 等于工作队列（改变 `GOMAXPROCS` 这个环境变量，可以改变分配数量）。*G* 是一个 goroutine，但是记住它只代表 goroutine 的当前状态，最明显的是它的程序计数器（PC）。G 相当于一个计算续体，使 Go 语言可以实现续体窃取。

在 Go 语言的运行时中，首先启动 M，然后是 P，最后是调度运行 G：



我发现对于我来说，很难分析这个算法是如何运作的，因为可使用的只有这几个符号，所以我将在这个分析中使用它们的全名。现在我们了解完了术语，让我们来看看 Go 语言的调度器是如何工作的！

正如我们所提到的，设置 GOMAXPROCS 可以控制运行时使用多少上下文。默认设置是主机上每个逻辑 CPU 分配一个上下文。与上下文不同的是，可能会有比 CPU 核心更多或更少的操作系统线程，帮助 Go 语言的运行时去管理 GC 和 goroutine 等。我提出这些是因为运行时拥有一个非常重要的保证：总会有足够的系统线程可以用来处理每个上下文。这使运行时可以进行一些重要的优化。运行时还包含当前未被使用线程的线程池。现在让我们来谈谈这些优化！

考虑一下如果任何一个 goroutine 被输入 / 输出或在 Go 语言的运行时之外进行的系统调用所阻塞，会发生什么。管理 goroutine 的系统线程也将被阻塞，并且无法继续执行或切换到任何其他的 goroutine。从逻辑上讲，这或许合理，但从性能的角度来看，Go 语言会进行更多的处理以尽可能让机器上的处理器保持活跃。在上述情况中，Go 语言会从系统线程中分离上下文，将上下文切换到另一个无阻塞的系统线程。这允许上下文调度那些后面可能要执行的 goroutine，使得运行时可以保持主机 CPU 的活跃。阻塞的 goroutine 仍然与阻塞的线程关联。

当 goroutine 阻塞最终结束的时候，主机系统线程会尝试使用一个其他系统线程来回退上下文，以便它可以继续执行先前被阻塞的 goroutine。但是，这并不总是能够顺利执行的。在这种情况下，线程将把它的 goroutine 放在全局上下文中，然后线程将进入休眠状态，并将其放入运行时的线程池中以供将来使用（例如，如果 goroutine 再次被阻塞）。

我们刚刚提到的全局上下文与我们之前对工作窃取算法在抽象层面上的讨论不太吻合。相差的这些内容，是 Go 语言在优化 CPU 利用率时所必须增加的实现细节。为了确保放在全局范围内的 goroutine 不会永久存在，我们需要在

工作窃取算法中添加一些额外的步骤。上下文会定期地检查全局上下文，以查看是否存在任何可以执行的 goroutine，并且当上下文的队列为空时，它在检查其他系统上下文之前，会先检查全局上下文，看是否可以进行工作窃取。

除了输入 / 输出和系统调用外，Go 语言还允许 goroutine 在任何函数调用期间被抢占。这与 Go 语言的设计哲学有关，即通过确保运行时优先进行细粒度的并发任务，来高效的调度任务。Go 语言的设计团队一直试图解决的一个重要的例外，就是那些既不执行输入 / 输出，也不进行系统调用或是函数调用的 goroutine。目前，这种类型的 goroutine 不能抢占，甚至会导致一些严重的问题，如 GC 的长时间等待甚至是死锁。幸运的是，从人们的反馈来看，这种情况发生的概率非常小。

向开发人员展示所有这些信息

现在你已了解 goroutine 背后的工作方式，让我们再次回过头来重申一下，开发人员与所有这些的连接点：关键字 go。仅此而已！

在函数或闭包之前敲上 go，你就会有一个会自动调度的任务，它将以最有效率的方式利用它所在的机器。作为开发者，我们依旧使用我们最熟悉的原语：function。我们不必理解新的处理方式，复杂的数据结构或调度算法。

可伸缩性、高效性，还有简单性。这就是 goroutine 的迷人之处。

尾声

现在我们遍历完 Go 语言中有关并发的全部内容：从第一原则，到基本用法，再到模式，以及运行时如何执行操作。我衷心希望本书已经让你很好的掌握了 Go 语言的并发特性，并帮助你出色的完成所有工作。谢谢！

附录 A

正如你在编写并发代码的过程中阐述的那样，你需要使用工具来编写程序并对其进行正确分析，以及一些有用的探针，以帮助你了解程序中发生的情况。幸运的是，Go 语言生态系统拥有来自 Go 语言团队和社区的丰富工具！本附录将讨论这些工具以及它们在开发之前，之中和之后如何为你提供帮助。由于本书侧重于并发性，因此我将限制该对话仅限于帮助你编写或分析并发代码的主题。我们还将简要介绍当 goroutine 发生 panic 时会发生什么。它并不经常发生，但是当你第一次看到它时，输出会有点混乱。

对 goroutine 异常的剖析

总有些事情无法避免：早晚你的程序会罢工的。如果上天眷顾你，那这个过程中不会有人或者计算机受到伤害，而最坏的情况是你将对着出错的栈信息发呆，不知从何下手。

在 Go 1.6 之前，当 goroutine 产生 panic 时，运行时会打印当前正在执行的所有 goroutine 的栈信息。有时候这会让判断故障原因更困难（至少会消耗一定时间）。在编写本书时，Go 1.6 及其更高的版本已经对程序的 panic 栈信息进行了简化。

例如，当执行这个简单的程序时：

```
1 package main
2
3 func main() {
4     waitForever := make(chan interface{})
5     go func() {
6         panic("test panic")
7     }()
8     <-waitForever
9 }
```

以下堆栈跟踪输出：

```
panic: test panic

goroutine 4 [running]:
main.main.func1() ❸
    /tmp/babel-3271QbD/go-src-32713Rn.go:6 +0x65  ❶
created by main.main
    /tmp/babel-3271QbD/go-src-32713Rn.go:7 +0x4e  ❷
exit status 2
```

❶ 指出 panic 产生的地方。

❷ 指出 goroutine 开始的地方。

❸ 指出 goroutine 运行的函数的名称。如果它是一个匿名函数（例如在当前程序中），则会自动分配一个唯一的标识符。

如果你希望看到程序崩溃时正在执行的所有 goroutines 的堆栈跟踪，那么可以通过将 GOTRACEBACK 环境变量设置为 all 来查看。

竞争检测

在 Go 1.1 中，为大多数命令增加了 -race 参数：

```
$ go test -race mypkg    # 对此 package 进行测试
$ go run -race mysrc.go  # 编译程序并运行
```

```
$ go build -race mycmd   # 构建此命令
$ go install -race mypkg # 安装此 package
```

如果你是一名开发人员，那么你需要的是一种可靠的方法来检测竞争状况，下面这些内容正是你需要的。使用竞争检测器有一点需要注意，就是该算法只能发现被执行代码中存在的竞争。由于这个原因，Go 语言团队建议在真实的负载中运行使用了 race 标记构建的应用程序。由于增加了执行更多代码的可能性，增加了可以发现竞争的概率。

你还可以通过调整一些环境变量来调整竞争检测器的行为，但通常情况下默认值已足够：

LOG_PATH

这告诉竞争检测器将日志写到 *LOG_PATH.pid* 文件中。你也可以传递特殊值给它：`stdout` 和 `stderr`。默认值是 `stderr`。

STRIP_PATH_PREFIX

这告诉竞争检测器在日志中删除文件路径的开头，以使它们更加简洁。

HISTORY_SIZE

这设置了每个 goroutine 历史记录的大小，它控制着每个 goroutine 历史记录的内存限制。值的有效范围是 [0，7]。`HISTORY_SIZE` 值从 0 开始，为 0 时会为 goroutine 历史记录分配 32 KB 内存，`HISTORY_SIZE` 最大为 7，分配 4 MB 内存。当你在日志中看到“failed to restore the stack”时，你需要调整这个参数的大小；不过，它会显著增加内存的消耗。

下面这个例子，我们曾第 1 章见过：

```
1 var data int
2 go func() { ❶
3     data++
4 }()
5 if data == 0{
6     fmt.Printf("the value is %v.\n", data)
7 }
```

你会收到这个错误：

```
==================
WARNING: DATA RACE
Write by goroutine 6:
  main.main.func1()
     /tmp/babel-10285ejY/go-src-10285GUP.go:6 +0x44 ❶

Previous read by main goroutine:
  main.main()
       /tmp/babel-10285ejY/go-src-10285GUP.go:7 +0x8e ❷

Goroutine 6 (running) created at:
  main.main()
      /tmp/babel-10285ejY/go-src-10285GUP.go:6 +0x80
==================
Found 1 data race(s)
exit status 66
```

❶ 表示 goroutine 试图进行非同步内存写入。

❷ 表示 goroutine（在这种情况下是 main goroutine）试图读取相同的内存。

竞争检测器是一个非常有用的工具，可用于自动检测代码中的竞态条件。我强烈建议将其整合为持续集成过程中的一部分。同样，由于竞争检测只能检测到已经产生的竞争，并且我们介绍过竞争条件有时难以触发的情况，因此应该在持续集成中运行真实环境的场景以尝试触发竞争。

pprof

在大型工程的代码中，有时难以确定程序在运行时的表现如何。有多少个 goroutine 正在运行？ CPU 是否被充分利用？内存使用情况如何？性能分析是解决这些问题的好方法，Go 语言在标准库中包含一个用于辅助分析的软件包，名为“pprof”。

pprof 是一个 Google 创造的工具，你可以在程序运行时显示当前的数据，或者使用这个工具来保存运行时的统计信息。这个程序的 help 标签可以帮助你很细致的描述了该程序的使用方法，所以在这里我们将只讨论 `runtime/pprof` 包，因为是它还涉及并发。

runtime/pprof 包非常简单，并具有预定义的配置文件，无需配置即可进行 hook 和显示：

```
goroutine    - stack traces of all current goroutines
heap         - a sampling of all heap allocations
threadcreate - stack traces that led to the creation of new OS threads
block        - stack traces that led to blocking on synchronization primitives
mutex        - stack traces of holders of contended mutexes
```

从并发的角度来看，其中大多数内容对于理解正在运行的程序中所发生的事情是很有用的。下面这个例子中有一个 goroutine，它可以帮助你理解 goroutine 泄漏：

```
log.SetFlags(log.Ltime | log.LUTC)
log.SetOutput(os.Stdout)

// 每 1s log 都会记录有多少个 goroutine 在并发执行
go func() {
    goroutines := pprof.Lookup("goroutine")
    for range time.Tick(1*time.Second) {
        log.Printf("goroutine count: %d\n", goroutines.Count())
    }
}()

// 创建一些永远不会退出的 goroutine
var blockForever chan struct{}
for i:=0; i < 10; i++ {
    go func() { <-blockForever }()
    time.Sleep(500*time.Millisecond)
}
```

这些内置的配置文件可以有效帮助你分析和诊断你的程序问题，当然你也可以自定义配置文件，以帮助你监控你的程序：

```
func newProfIfNotDef(name string) *pprof.Profile {
    prof := pprof.Lookup(name)
    if prof == nil {
        prof = pprof.NewProfile(name)
    }
    return prof
}

prof := newProfIfNotDef("my_package_namespace")
```

作者介绍

Katherine Cox-Buday 是一名计算机科学家，目前就职于 Simple online banking。她的业余爱好包括软件工程、创作、Go 语言（igo、baduk、weiquei）和音乐，这些都是她长期的追求，并且有着不同层面的贡献。

封面介绍

本书封面动物是一只短耳朵的象鼩（Macroscelides proboscideus），它是一种小型哺乳动物，原产于纳米比亚、博茨瓦纳和南非的干旱地区。短耳象鼩被称为“sengi”，其名字来源于它们特别像大象鼻子一样的细长鼻子。

短耳象鼩的体重在 28~43g，长约 10cm，成为象鼩家族中的最小物种。它们的皮毛是棕色和灰色的，腹部是白色的。它们以昆虫为食，像白蚁、蚂蚁、蠕虫，还有像浆果和树叶芽一样的植物。

虽然短耳象鼩主要是独居动物，但它们是少数几种一夫一妻制的哺乳动物之一。交配时将会联合起来保护它们的领地不受其他动物的伤害。一只短耳朵象鼩的寿命在野外是 1~2 年，但它们被证明可以活到 4 年。

O’Reilly 的许多书的封面动物都濒临灭绝，所有这些动物对世界都很重要。想了解更多关于如何帮助它们的信息，请登录 *animals.oreilly.com*。

封面图片来自 Braukhaus Lexicon。

O'REILLY®

Go语言并发之道

众所周知，构建正确的并发程序并不容易，但幸运的是，Go语言这门开源编程语言使编写并发程序变得简单并易于处理。如果你是熟悉Go语言的开发者，这本书非常实用，它展示了在你系统中融入并发特性的最佳实践和模式。

本书作者带你一步一步深入这些方法。你将理解 Go语言为何选定这些并发模型，这些模型又会带来什么问题，以及你如何组合利用这些模型中的原语去解决问题。学习那些让你在独立且自信的编写与实现任何规模并发系统时所需要用到的技巧和工具。

- 理解Go语言如何解决并发难以编写正确这一根本问题。
- 学习并发与并行的关键性区别。
- 深入到Go语言的内存同步原语。
- 利用这些模式中的原语编写可维护的并发代码。
- 将模式组合成为一系列的实践，使你能够编写大规模的分布式系统。
- 学习 goroutine 背后的复杂性，以及Go语言的运行时如何将所有东西连接在一起。

Katherine Cox-Buday是一名计算机科学家，目前工作于Simple online banking。她的业余爱好包括软件工程、创作、Go 语言，以及音乐，这些都是她长期的追求，并且有着不同层面的贡献。

"Katherine有深度且有风格的用Go语言解决并发的复杂性。她沉着地描述了从原语到模式的所有内容。编写任何用于生产的并发代码之前都应当阅读这本书。"

——**Brian Ketelsen**
Microsoft，
Organizer,GopherCon

"Go语言支持并发是一个伟大语言优势，Katherine对关键原则给出了通俗易懂的解释，她用非常棒的例子展示了并发的实践。每一个Go语言的开发者都应当阅读这本书！"

——**Liz Rice**
Chief Technology Evangelist，
Aqua Security

O'Reilly Media, Inc.授权中国电力出版社出版

ISBN 978-7-5198-2494-5

定价：58.00元